UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VINICIUS JOSÉ DE SIQUEIRA

History-based Prioritization in the Context of Manual Testing: *a Study in a Real Industrial Setting*

Recife

2022

VINICIUS JOSÉ DE SIQUEIRA

History-based Prioritization in the Context of Manual Testing: *a Study in a Real Industrial Setting*

Trabalho apresentado ao Programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador (a): Breno Alexandro Ferreira de Miranda

Recife

2022

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S618h Siqueira, Vinicius José de
History-based prioritization in the context of manual testing: a study in a real industrial setting / Vinicius José de Siqueira. – 2022.
79 f.: il., fig., tab.

Orientador: Breno Alexandro Ferreira de Miranda.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022

Inclui referências.

1. Engenharia de software. 2. Teste de regressão. I. Miranda, Breno Alexandro Ferreira de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2022-167

Vinicius José de Siqueira

“History-based Prioritization in the Context of Manual Testing: a Study in a Real Industrial Setting”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 19/09/2022.

BANCA EXAMINADORA

_____

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática / UFPE

_____

Profa. Dra. Genaína Nunes Rodrigues
Departamento de Ciência da Computação / UnB

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE
(**Orientador**)

To my family and close friends who have been cheering me on.

ACKNOWLEDGEMENTS

I want to open these acknowledgments by dedicating the superior forces that always guide me, whether, in the quiet moments or the most turbulent ones, these forces always keep me on the axis, focused and running after my goals.

I would like even more to leave a special thanks to my advisor, Breno Miranda, an incredible person who appeared on my timeline during a coffee break in the pantry at work, talking precisely about one of the goals I had in mind and that had not yet happened, that was my desire to apply for the master's degree. Once again, the superior force put this incredible human being in my path; thank you, professor, for all the teachings and great patience that you have to share all your knowledge; if it wasn't for you guiding me, I think this work would not have left my mind and from the paper.

I also cannot fail to thank my wife, Wêdja Kátia, for having understood my moments of absence during this master's period, she encouraged me to keep moving forward at various times.

I could not fail to thank my dear mother, Hilda Benicio, and my queen who is my grandmother, Maria Benicio - *In memoriam*, who have always guided me along the paths of knowledge, if it weren't for them, many of my achievements that I have in my life (all coming from the studies) would not be possible. You can be sure that I will be forever grateful for everything you have given me.

I would also like to thank my co-workers, who encouraged me and also supported me in some moments when I needed to be more absent from work to be able to dedicate myself a little more in some moments of the master's, thank you Vitor Lima, Jully Quintans, Emanuel Oliveira, Lucas Carneiro, and also to Mr. Eliot Maia whom I had the honor to count on to help with the review of part of the work and for having passed on some feedbacks and insights.

Finally, I would like to thank all my family and close friends who have also been cheering for me all this time and who were able to help directly or indirectly during this period; now, we will be able to see each other a little more and celebrate, *rs*.

"Program testing can be used to show the presence of bugs, but never to show their absence." (DIJKSTRA et al., 1970)

ABSTRACT

Many test case prioritization techniques have been proposed with the ultimate goal of speeding up fault detection. History-based prioritization, in particular, has been shown to be an effective strategy. However, most empirical studies on this topic have focused on the context of automated testing. Investigating the effectiveness of history-based prioritization in the context of manual testing is important because, despite the popularity of automated approaches, manual testing is still largely adopted in the industry. In this work, we propose two history-based prioritization heuristics and evaluate them in the context of manual testing in a real industrial setting. We compared our proposed approaches against alternative prioritization strategies, including a state-of-the-art history-based approach, an optimal prioritization, the real ordering followed by the testers, the ordering suggested by a test management tool, and a random ordering. For our evaluation, we collected historical test execution information from 35 products, spanning over seven years of historical information, accounting for a total of 3,196 unique test cases and 5,859,989 test results. The results of our experiments using historical test execution data from real subjects and with real faults showed that the effectiveness of the proposed approaches is not far from a theoretical optimal prioritization and that they are significantly better than alternative orderings of the test suite, including the state-of-the-art history-based approach, and the execution order followed by the testers during the real execution of the test suites evaluated as part of our study. With respect to efficiency, our proposed approaches yield similar results and they are both better (faster) than the state-of-the-art history-based competitor.

Keywords: regression testing; test case prioritization; history-based prioritization; manual testing.

RESUMO

Muitas técnicas de priorização de casos de teste foram propostas com o objetivo final de acelerar a detecção de falhas. A priorização baseada em histórico, em particular, tem se mostrado uma estratégia eficaz. A maioria dos estudos empíricos realizados neste tópico, no entanto, se concentraram no contexto de testes automatizados. Investigar a eficácia da priorização baseada em histórico no contexto de testes manuais é importante porque, apesar da popularidade das abordagens automatizadas, o teste manual ainda é amplamente adotado na indústria. Neste trabalho nós propomos duas heurísticas de priorização baseadas em histórico e as avaliamos no contexto de testes manuais em um ambiente industrial real. Nós comparamos nossas abordagens propostas com estratégias alternativas de priorização, incluindo a abordagem do estado da arte baseada em histórico, uma priorização ótima, a ordenação real seguida pelos testadores, a ordenação sugerida pela ferramenta de gerenciamento de teste e uma ordenação aleatória. Para nossa avaliação nós coletamos informações históricas de execução de testes para 35 produtos, abrangendo mais de sete anos de informações históricas, contabilizando um total de 3.196 casos de teste únicos e 5.859.989 resultados de teste passados. Os resultados de nossos experimentos mostraram que a eficácia das abordagens propostas não estão longe de uma teórica priorização ótima, e que são significativamente melhores do que as alternativas de ordenações das suítes de testes, incluindo a abordagem utilizada como comparação do estado da arte, a ordem sugerida pela ferramenta de gerenciamento de testes e a ordem de execução seguida pelos testadores durante a execução real das suítes de testes avaliadas durante o nosso estudo. Com relação à eficiência, nossas abordagens propostas produzem resultados semelhantes e ambas são melhores (mais rápidas) do que a abordagem concorrente do estado da arte baseada em histórico.

Palavras-chaves: teste de regressão; priorização de casos de teste; priorização baseada no histórico; testes manuais.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

APFD	Average Percentage of Fault Detected
CR	Change Request
OS	Operational System
PO	Product Owner
QA	Quality Assurance
RT	Regression Testing
SME	Subject Matter Expert
SUT	Software Under Test
TCP	Test Case Prioritization
TCS	Test Case Selection
TM	Test Manager
TSM	Test Suite Minimization
TSR	Test Suite Reduction
TTC	The Testing Company

CONTENTS

1 INTRODUCTION

1.1 CONTEXT

Regression testing is an important testing technique that aims at verifying that modifications in the software under test (SUT) have not introduced new faults — or reintroduced previously-known faults. In other words, in regression testing the aim is to identify possible regressions (i.e., previously-working features do not work properly in the latest version). As the software evolves, however, the size of the regression test suite tends to grow, making regression testing an expensive technique. To make regression testing more cost-effective, many regression techniques have been proposed in the literature, including test case selection, test suite reduction (or minimization), and test case prioritization (ENGSTRÖM; RUNESON; SKOGLUND, 2010; YOO; HARMAN, 2012; CATAL; MISHRA, 2013; KHAN et al., 2018).

Test case prioritization aims at ordering a test suite in such a way that a given reward function is maximized. Generally, the reward function that one wants to maximize is the speed at which faults are revealed (i.e., the sooner faults are revealed, the better). Because it is not possible to know *a priori* which test cases would reveal which faults, many test case prioritization techniques rely on proxies to choose which test cases should be run first. These proxies could be code coverage (NARDO et al., 2015; MIRANDA; BERTOLINO, 2017) (which parts of the code are exercised by this test?), similarity (MIRANDA et al., 2018; GRECA et al., 2022) (how (dis)similar is this test with respect to the set of already-executed tests?), history (KIM; PORTER, 2002; FAZLALIZADEH et al., 2009; HAGHIGHATKHAH et al., 2018; BERTOLINO et al., 2020) (how did this test perform in previous executions?), among others.

History-based Prioritization, in particular, has been shown to be an effective strategy for ordering regression test suites. Most studies on history-based prioritization, however, have focused on the context of automated testing (KIM; PORTER, 2002; FAZLALIZADEH et al., 2009; HUANG; PENG; HUANG, 2012; NOOR; HEMMATI, 2015; HAGHIGHATKHAH et al., 2018), with little attention being paid to the context of manual testing. Investigating the effectiveness of history-based prioritization in the context of manual testing is an important question because, despite the popularity of automated approaches, manual testing is still largely adopted in industry (HAAS et al., 2021).

To investigate this question, we developed two history-based prioritization heuristics and evaluated them in the context of manual testing in a real industrial setting. The company's name cannot be disclosed due to a non-disclosure agreement, and hereafter we refer to it as TTC (short for "the testing company").

TTC works mainly with black-box tests that are run manually and one important step in TTC's processes is the creation of test suites. For creating the test suites, the *test manager* considers all the relevant product-related information (e.g., category, features, available applications) and defines which test sets will be used to verify the product. Given that TTC is a large company, several products are tested in parallel and the number of tests to be performed can run into the thousands. The test suites created are passed on to the testing team with no specific execution order and it is left for the tester to decide in which order the test cases should be executed. In this work we investigate the use of history-based prioritization approaches in the context of TTC.

For our evaluation, we collected historical test execution information from 35 products, spanning over seven years of historical information, accounting for a total of 3,196 unique test cases and 5,859,989 test results. We then assessed the performance of our approach with respect to alternative orderings of the test suite: an optimal prioritization; a state-of-the-art history-based prioritization (HAGHIGHATKHAH et al., 2018); the real ordering followed by the testers; the ordering suggested by the TTC test management tool; and a random ordering.

The main contributions of this work are:

- the proposal of two history-based prioritization heuristics inspired by the context of manual testing.
- an empirical evaluation of history-based prioritization in the context of manual testing in a real industrial setting. For our evaluation, we collected historical test execution information from 79 test suites created to validate 35 products of two different families lines, spanning over seven years of historical information, accounting for a total of 3,196 unique test cases and 5,859,989 test results.
- the results of our experiments using historical test execution data from real subjects and with real faults: the results showed that the effectiveness of the proposed approaches, measured by their APFD (Family-dependent = 0.81 and Family-independent = 0.77), is not far from a theoretical optimal prioritization (0.99) and that they are significantly better than alternative orderings of the test suite, including the state-of-the-art history-

based approach (0.72), and the execution order followed by the testers during the real execution of the test suites (0.52), evaluated as part of our study.

1.2 DISSERTATION ORGANIZATION

This section describes how the subsequent chapters of this work are structured. The content of Chapters 4 and 5 were partially published at the 48th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA'22) (SIQUEIRA; MIRANDA, 2022a):

- **Chapter 2:** Introduces basic concepts and definitions related to this dissertation. We present software testing levels, types, and regression testing techniques. We also provide a brief discussion on manual testing and cover some concepts that are related to the industrial context where our study was carried out.
- **Chapter 3:** Discusses previous work in the literature related to regression testing and its techniques with a particular focus on history-based prioritization and manual testing.
- **Chapter 4:** Proposes the adoption of simple, yet effective and efficient, history-based prioritization heuristics by leveraging the large database of historical test results available at the company where we carried out our study.
- **Chapter 5:** Presents the experimental study that evaluates the effectiveness and efficiency of the proposed heuristics. In this chapter, we provide the goal definitions, followed by the planning, preparation, and analysis of our study. We present the threats to the validity of our study and discuss the experimental results.
- **Chapter 6:** Concludes our work and discusses our approaches' limitations, contributions, and future work.

2 BACKGROUND

This chapter aims to introduce basic concepts and definitions related to this work. First, we present Software Testing and its levels, types, and techniques, then the Regression Testing concepts and its techniques, and at the end of the chapter, the Manual Testing definitions and a few concepts used by the testing company. These concepts will be helpful to have a better understanding of the rest of this dissertation.

2.1 SOFTWARE TESTING

The advent of new technologies and the rise of existing ones require more reliable and better quality software year by year, making the software increasingly complex and challenging to test. Still, if it is not adequately tested, errors may pass from the development environment to the production environment. Once in production, correcting these errors can become very expensive for the company. Therefore, software testing can be considered an activity to ensure that the software's functionalities and quality are acceptable. It is also the main activity that reveals faults.

The quality of software can be guaranteed through the software testing process, but usually, software testing and validation phases are often expensive and time-consuming processes (SRIKANTH; COHEN; QU, 2009; MYERS; SANDLER; BADGETT, 2011; DO, 2016).

2.1.1 Test Levels

In general, software testing is divided into four main categories, which are: *Component Testing*, *Integration Testing*, *System Testing*, and *Acceptance Testing*.

2.1.1.1 Component Testing

Component Testing, also known as *Unit Testing*, is the lowest stage of the testing scale being applied to the smallest code components created, to ensure that they meet the specifications, they aim to try to find defects in modules, objects, classes, code snippets, specific applications, or other units that are individually tested. They are usually derived from the component specifications or the software design itself. To create these tests, it is necessary to have

access to the code being tested and to have some support in the development environment, such as a unit test framework or a debugging tool. In practice, defects found in this phase are corrected as soon as they are discovered.

Considering the characteristics and functionality, component tests verify the functioning of a part of the system or software in isolation, and in most cases, these tests are carried out by the developers themselves (RIOS; MOREIRA, 2006).

2.1.1.2 Integration Testing

The main purpose of Integration Testing is to test interfaces between components, interactions with different parts of the system, and interfaces between systems. At each stage of integration, the testers focus on testing the communication between these modules and not the individual functions of each component to verify that together they work correctly, that is, to ensure that the interfaces work and that the data are being processed correctly according to specifications.

Integration tests can run incrementally as each module or component is added. This type of testing should preferably be performed by the development team or by a testing team.

2.1.1.3 System Testing

System Testing are the tests performed when the system is complete and already integrated to check the system's compliance with the requirements that have been specified (IEEE, 2008). They aim at the execution of the system as a whole, usually in a controlled operating environment, to validate the correctness of the implementation of its functions. In system tests, the system operation is simulated, passing through all functions as close as possible to what can or will occur in the production environment.

In this stage, load, performance, usability, compatibility, security, and recovery tests are carried out, among others.

The testing team performs these tests, but they do not replace acceptance tests performed by users.

2.1.1.4 Acceptance Testing

Acceptance Testing are the tests of the final phase of system execution, usually carried out by sponsors/users intending to verify that the solution meets the business objectives and requirements before using it in the production environment. The primary purpose of acceptance testing is to establish the reliability of the system, part of the system, or specific non-functional characteristics. This type of testing can happen at various times in the software lifecycle, such as the acceptance test of new functionality. This type of test can also be referred to as *Validation Testing* (IEEE, 2008).

Although the users perform these tests, they are conducted with the support of the test team and the project team.

2.1.2 Test Types

2.1.2.1 Functional Testing

Functional Testing is performed to assess the system's conformance to specified functional requirements. Generally, these tests are indicated to detect interface-related and behavior-related errors. They can be applied in all test phases: unit, integration, system, and acceptance. There are several types of functional tests. To mention just a few: *Smoke Test*, *Sanity Test*, *Regression Test*, and *Usability Test*.

2.1.2.2 Non-Functional Testing

Non-functional Testing are tests related to the use of the application, taking into account performance, usability, reliability, security, availability, maintainability, and other features involved. These are generally not tests requested by the sponsors, but they are still necessary to guarantee the minimum characteristics of quality software.

2.1.2.3 Structural Testing

Structural Testing, or *white-box* testing, are tests developed taking into account the system's internal structure (at the coding level), allowing a more accurate verification of the

software's operation. They are designed by analyzing the source code and writing test cases covering the software component's functionality. This technique is seen as a complementary technique to the functional technique. This type of testing is recommended for the unit testing and integration testing phases, being the responsibility of the system developers, as they know well the code produced.

2.1.3 Testing Techniques

2.1.3.1 Black-Box

The Black-Box tests have this name for the simple fact that we do not have detailed information about what happens at the coding level, that is, without being able to see what is "inside the box", they aim to verify the functionality and the adherence to requirements, taking into account the external perspective of the user, without relying on any knowledge of the code or internal logic of the tested component.

2.1.3.2 White-Box

The White-Box tests, in turn, allow us to access what is "inside the box" and they aim to evaluate the code statements, the internal logic of the coded component, the settings, and other elements. However, to carry out these tests, we need to understand the internal perspective of the system, as mentioned above, and have enough programming knowledge.

2.2 REGRESSION TESTING

Regression Testing (RT) is an important testing technique that aims at verifying that modifications in the Software Under Test (SUT) have not introduced new faults — or reintroduced previously known faults at parts of the code that were working correctly before the modifications. As the software evolves, however, the size of the regression test suite tends to grow, making regression testing an expensive technique.

As this technique is applied to each new change made to the software, a set of data and information is generally maintained as a baseline. Each new integration is compared with this baseline to verify that changes introduced later will not damage code already considered good

and that has already been accepted.

To make regression testing more cost-effective, many regression techniques have been proposed in the literature, including Test Case Selection (TCS), Test Suite Reduction (TSR) (or Test Suite Minimization (TSM)), and Test Case Prioritization (TCP) (ENGSTRÖM; RUNESON; SKOGLUND, 2010; YOO; HARMAN, 2012; CATAL; MISHRA, 2013; KHAN et al., 2018).

Test case prioritization aims at ordering a test suite in such a way that a given reward function is maximized. Generally, the reward function that one wants to maximize is the speed at which faults are revealed (i.e., the sooner faults are revealed, the better). Because it is not possible to know *a priori* which test cases would reveal which faults, many TCP techniques rely on proxies to choose which test cases should run first. These proxies could be code coverage (NARDO et al., 2015; MIRANDA; BERTOLINO, 2017) (which parts of the code are exercised by this test?), similarity (MIRANDA et al., 2018; GRECA et al., 2022) (how (dis)similar is this test with respect to the set of already-executed tests?), history (KIM; PORTER, 2002; FAZLALIZADEH et al., 2009; HAGHIGHATKHAH et al., 2018; BERTOLINO et al., 2020) (how did this test perform in previous executions?), among others.

History-based Prioritization, in particular, has been shown to be an effective strategy for ordering regression test suites (KIM; PORTER, 2002). Most studies on history-based prioritization, however, have focused on the context of automated testing (KIM; PORTER, 2002; FAZLALIZADEH et al., 2009; HUANG; PENG; HUANG, 2012; NOOR; HEMMATI, 2015; HAGHIGHATKHAH et al., 2018), with little attention being paid to the context of manual testing. Investigating the effectiveness of history-based prioritization in the context of manual testing is an important question because, despite the popularity of automated approaches, manual testing is still largely adopted in industry (HAAS et al., 2021).

2.2.1 Regression Testing Techniques

The crying out for efficient regression testing strategies is thus becoming more and more critical. So, by keeping in mind the primary purpose of regression testing and trying to minimize the expenses mentioned above, it is possible to use at least three different Regression Testing Techniques to optimize the execution of our test suites: *Test Suite Minimization*, *Test Case Selection* and *Test Case Prioritization* (YOO; HARMAN, 2012; MUKHERJEE; PATNAIK, 2018).

2.2.1.1 *Test Suite Minimization*

Test Suite Minimization, or Test Suite Reduction, is one of the three known techniques used to use available resources better when performing test suites. In the case of TSM, we can say that its main objective is to minimize or reduce the suites by removing test cases that are identified as obsolete cases or cases that are considered redundant for the execution of this suite, following some criteria that will not compromise its effectiveness in the future, such as fault-detection capability, to create a minimally representative of the original suite (JEFFREY; GUPTA, 2007; KHAN et al., 2018; CRUCIANI et al., 2019; BAJAJ; SANGWAN, 2019).

2.2.1.2 *Test Case Selection*

In Test Case Selection, unlike TSM, where the objective is to reduce the suite by removing cases, the main focus is to select the test cases that are important to cover a specific region of the SUT, creating a subset of test cases following some specific criteria, for example, to test only the areas of the code where there has been some modification. But in the TCS technique, it is crucial to consider the trade-off between rerunning all test cases and the risk of losing the opportunity to get faults that might be introduced by the side effects of changes made to the software. It is also necessary to take into account that the suite created after the application of the TCS technique must detect as much as possible the faults found by the original suite (ELBAUM et al., 2003; ENGSTRÖM; RUNESON; SKOGLUND, 2010; BERTOLINO et al., 2019).

2.2.1.3 *Test Case Prioritization*

Test Case Prioritization, on the other hand, aims to order the test cases based on some properties or heuristics so that the test cases that are at the top of the rank are executed as a priority, with the intention that its output produces a previously planned result, which in the majority of times is finding faults as early as possible. This reordering of test cases through prioritization can make the suite demonstrate a better efficiency of its results. It is worth noting that while the TSM and TCS techniques reduce the original test suite, the TCP technique only reorders the suite to achieve a better result, but without removing any of the test cases (YOO; HARMAN, 2012; HAO; ZHANG; MEI, 2016; BAJAJ; SANGWAN, 2019).

To define these heuristics, a few different TCP techniques can be used, some of these techniques were classified as *Cost-aware, Coverage-based, Distribution-based, History-based, Human-based, Model-based, Probabilistic,* and *Requirement-based* (YOO; HARMAN, 2012).

The History-based Technique

In the history-based prioritization approach, what is used as input to define the test case prioritization is the historical data of past executions. As an example, it can use the number of past executions test results that's revealed a fault by a test case and use this information as a weight during the analysis (KIM; PORTER, 2002).

Evaluation Metric

Independently of the approach proposed for a test case prioritization, it is essential to use some measurement metric to determine their effectiveness. This is important to estimate the efficacy of the TCP approach proposed as well to benchmark the effectiveness of this approach against other approaches.

According to (HAO; ZHANG; MEI, 2016; KHATIBSYARBINI et al., 2018; MUKHERJEE; PATNAIK, 2018), most TCP techniques use the Average Percentage of Fault Detected (APFD) to evaluate its effectiveness. This measurement metric was presented initially by (ROTHERMEL et al., 1999) TCP research. It formally can be represented by the Equation 2.1, where m represents the number of faults found in a test suite, n represents the total number of test cases, and TF_j represents the first test case in the prioritized test suite that detects the fault j.

$$APFD = 1 - \frac{\sum_{j=0}^{m} TF_j}{nm} + \frac{1}{2n} \tag{2.1}$$

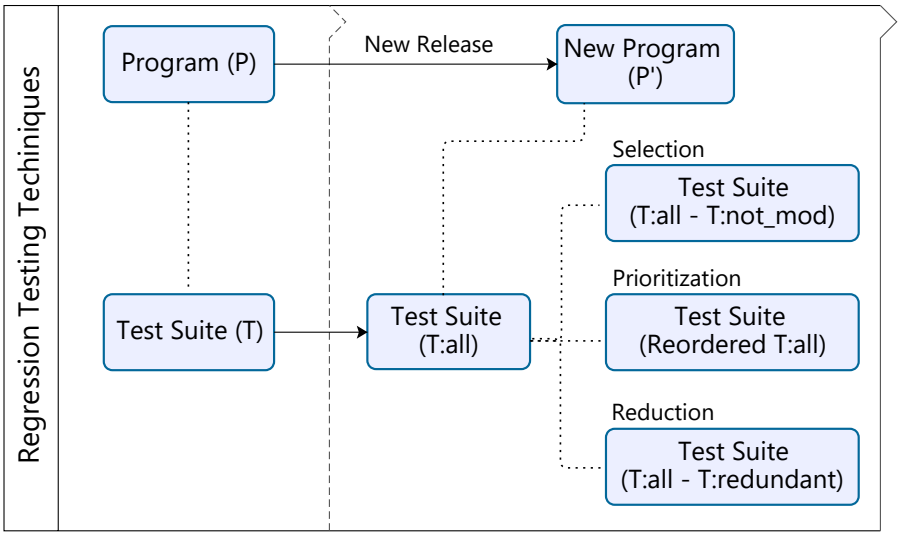
The APFD value will vary between 0 to 1, where the higher this value is, the better will be the fault detection of the prioritized test suite. So a higher APFD can appoint that the faults are found faster using fewer cases. As mentioned above, this measurement is widely used to evaluate the effectiveness of TCP.

While the TSM and TCS techniques focus on reducing the number of test cases, the first removing obsolete or redundant cases and the second selecting cases that are considered most

important, it might be possible that crucial test cases are left out of the test suite, while in the technique of TCP all test cases are kept, but re-ordering the sequence in which these cases need to be executed, that is, all cases will be considered in the prioritization.

In Figure 1, it is possible to see how each technique behaves, where P is our program that was updated generating a new release P', and T is our test suite that is used to test the program P. Taking into account that instead of using the test suite T exactly as it is, we are going to use one of the three techniques seen above, which are exemplified on the right side of Figure 1. The TCS technique selects a subset of tests from the $T : all$ suite and decreases the cases from it that cover fundamental parts of the $T : not\_mod$ program. The TSM technique reduces the size of the suite by removing from $T : all$ all cases that are considered redundant $T : redundant$, and the TCP technique reorders all $T : all$ cases with the intent to obtain better effectiveness of the test suite, such as increasing the failure detection rate (DO, 2016).

Figure 1 – Regression Testing Techniques



Source: Adapted by the author from (DO, 2016)

Yoo and Harman (YOO; HARMAN, 2012), broadly cover these three main test optimization techniques, as well as systematic reviews in the literature that bring a good overview of studies that have been conducted in this area of Regression Testing: Test Suite Minimization (KHAN et al., 2018), Test Case Selection (ENGSTRÖM; RUNESON; SKOGLUND, 2010) and Test Case Prioritization (KHATIBSYARBINI et al., 2018).

2.3 MANUAL TESTING

Manual Testing, as the name suggests, is the process of manually testing software to find defects (faults, bugs) in that software. This type of testing requires the tester to play the end-user role, using most of the software's functionality to ensure that the software is achieving the correct behavior. To perform these tests, the tester follows a *Test Plan* that leads him through a set of *Test Cases*, which in turn have a set of steps and expected results.

The advantage of using manual tests is the possibility of finding more defects since the human eyes observe and judge better than automated tools.

According to the *IEEE Standard for Software and System Test Documentation* (IEEE, 2017), a systematic approach focuses on predetermined test cases and usually involves the following steps:

- Choosing a high-level test plan where a general methodology is chosen and resources such as people, computers, and software licenses are identified and acquired.
- The writing of detailed test cases, identifying clear and concise steps to be taken by the tester with the expected results.
- Assigning test cases to testers (test case allocation), who will manually follow the steps and record the results.
- And lastly, creating a test report detailing what the testers found. From the results, managers will use the report to determine whether the software can be released or not; otherwise, this same report is used by engineers to identify and correct problems that have been reported.

2.3.1 TTC Concepts and Test Artefacts

The purpose of this section is to present some terms/concepts that The Testing Company (TTC) uses in addition to those previously introduced and that can be referenced in the course of this dissertation. Some of these terms have the same meaning as commonly used in the software testing literature, but others may have a slightly different meaning.

2.3.1.1 Testing Artifacts

- **Test Plan:** The Test Plan is a document that describes the technical and management approach to be followed for testing a system or component. Among the items identified, we can mention: the scope, approach, resources, schedule of intended test activities, test items, features to be tested, testing tasks, responsibilities, required resources for the testing activity, and risks requiring contingency planning (IEEE, 2017).

- **Test Suite:** The Test Suite is a collection of test cases that have been defined to test a software program, which serves to validate the behavior of this software program. A test suite usually contains detailed instructions or objectives for each collection of test cases and information about the system configuration to be used during testing. A test case group can also include prerequisite states or steps and descriptions of the tests to follow. Test case collections are sometimes incorrectly called as Test Plan, especially in industrial environments.

- **Test Case:** The Test Case is a *triple* formed by: test inputs, execution conditions, and expected results. The test case is developed for a particular objective, such as to exercise a particular program path, to check some behavior, or to verify compliance with a specific requirement (IEEE, 2017). A test case also identifies constraints on the test procedures resulting from using that specific test case. The test cases are separated from test designs to allow for use in more than one design and reuse in other situations (IEEE, 1983).

- **Test Report:** Following the *IEEE Standard for Software Test Documentation* (IEEE, 1983), the Test Report should be covered by four document types:

    1. A test item transmittal report, which identifies the test items being transmitted for testing if separate development and test teams are involved or if a formal beginning of test execution is desired.
    2. A test log, which the test team uses to record what occurred during test execution.
    3. A test incident report, which describes any event during the test execution requiring further investigation.
    4. A test summary report, that summarizes the testing activities associated with one or more test-design specifications.

- **Change Request (Defect Report):** A Change Request (CR) (the term common used on the TTC), or a Defect Report as known in the literature, is a document with details about what steps should be followed to identify defects, what actions make the defects appear, and what are the expected results instead of the application showing error (defect) while taking particular step-by-step actions.

  Change requests are usually created by the test team and sometimes for the end-users (suppose the company does not have a robust software test team. In this case, it will be easier for the users to detect more defects and report them to the support team of the software development since the majority of the users curiously try out every feature in the application). These change requests are created to help developers find out the defects easily and fix them up.

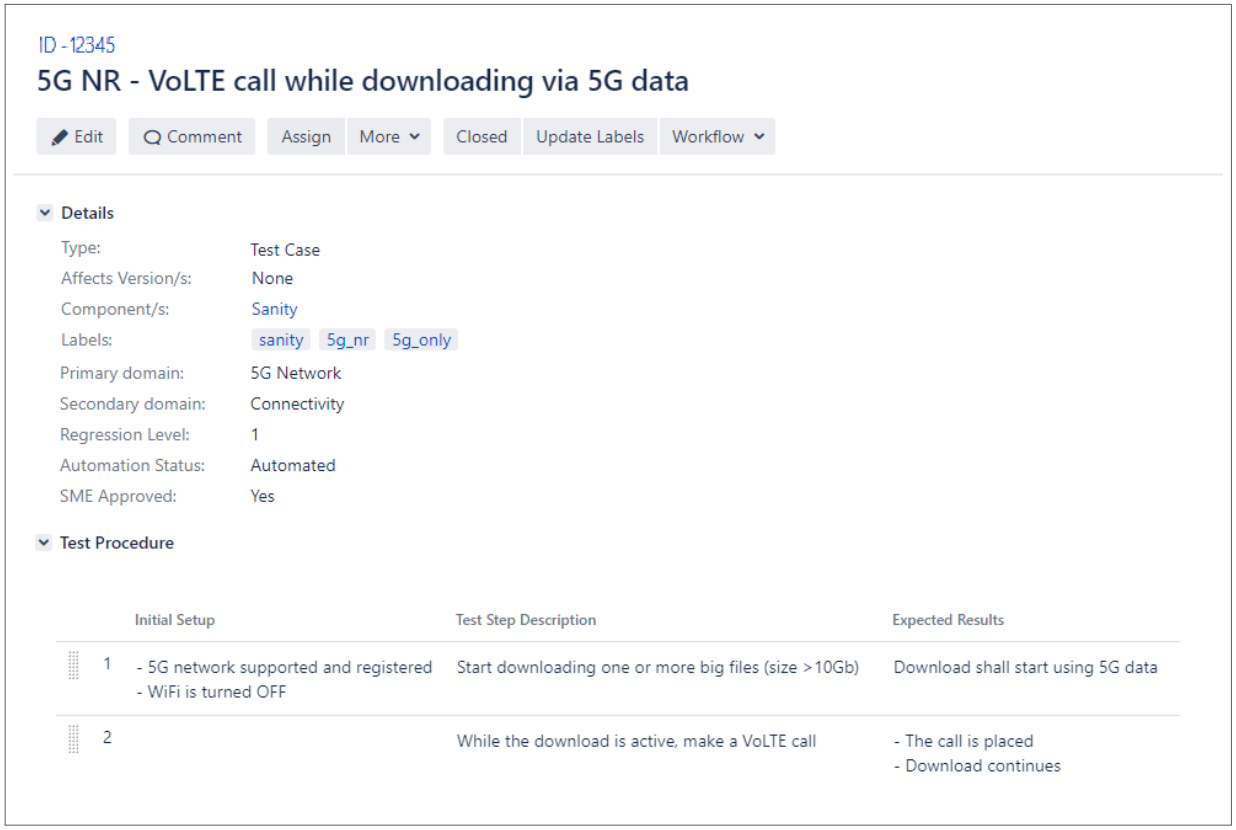
  Change requests should be short, organized, straight to the point, and cover all the information that the development team needs to know to detect the actual defects in the report by doing what and how the report describes. It is usual for developer teams to get defect reports from the testers that are either too short or too long to understand what went wrong.

2.3.1.2 Test Case Details

A Test Case is composed of essential information such as the identifier, the summary (a brief description that defines the main objective of the test), and a sequence of inputs, conditions, and expected results. As a good standard for writing these test cases, ideally, each condition has its expected result. In addition to this triple of information, test cases can also include other information such as: *Components, Labels, Feature ID, Primary domain, Secondary domain, Regression Level, Automation Status, etc*. In Figure 2, we have illustrated a structure of a test case very close to a test case used by TTC.

Figure 2 shows a test case whose objective is to make a voice call using the 4G network while data is being downloaded using the 5G network. In the first step, we have the precondition that the WiFi is off, and the tester is asked to download a large file, preferably with more than 10GB, and the expected result is to ensure that this download is being done over the 5G network. Meanwhile, it is requested in the next step to make a voice call using the 4G mobile communication network, in which case the tester needs to verify that the voice call is made

Figure 2 – Test Case example



Source: Adapted by the author

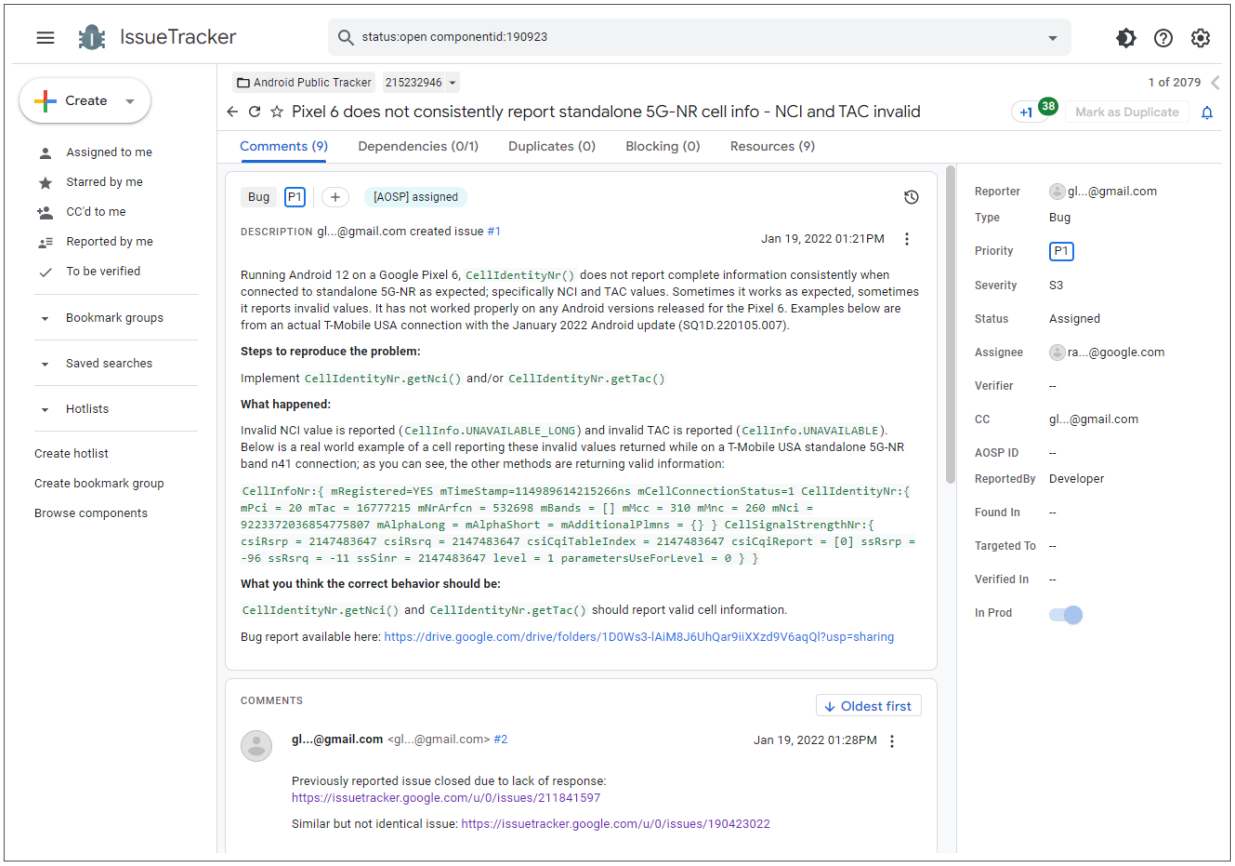
without problems and that the download will continue.

2.3.1.3 Change Request Details

The CR has a structure very similar to the design of the test case as in the TTC they use the same software used to store the test cases but in a different instance. In the CR, we have the identifier, the summary, and other predefined fields that may or may not be used during the creation of this CR; among these fields, we can highlight some such as *affected product, software version, labels, description (free-form field), attachments, watchers, comments* and more. Figure 3 shows an example of a defect stored in Google's Buganizer[1], however as TTC does not use buganizer as its primary defect reporting tool, it uses proprietary software, this buganizer example anyway illustrates well the approach used by TTC without losing information.

[1] https://issuetracker.google.com/issues

Figure 3 – Google Issue Tracker example



Source: https://issuetracker.google.com

Other fields that are also very important are the follow-up fields of this CR, which are:

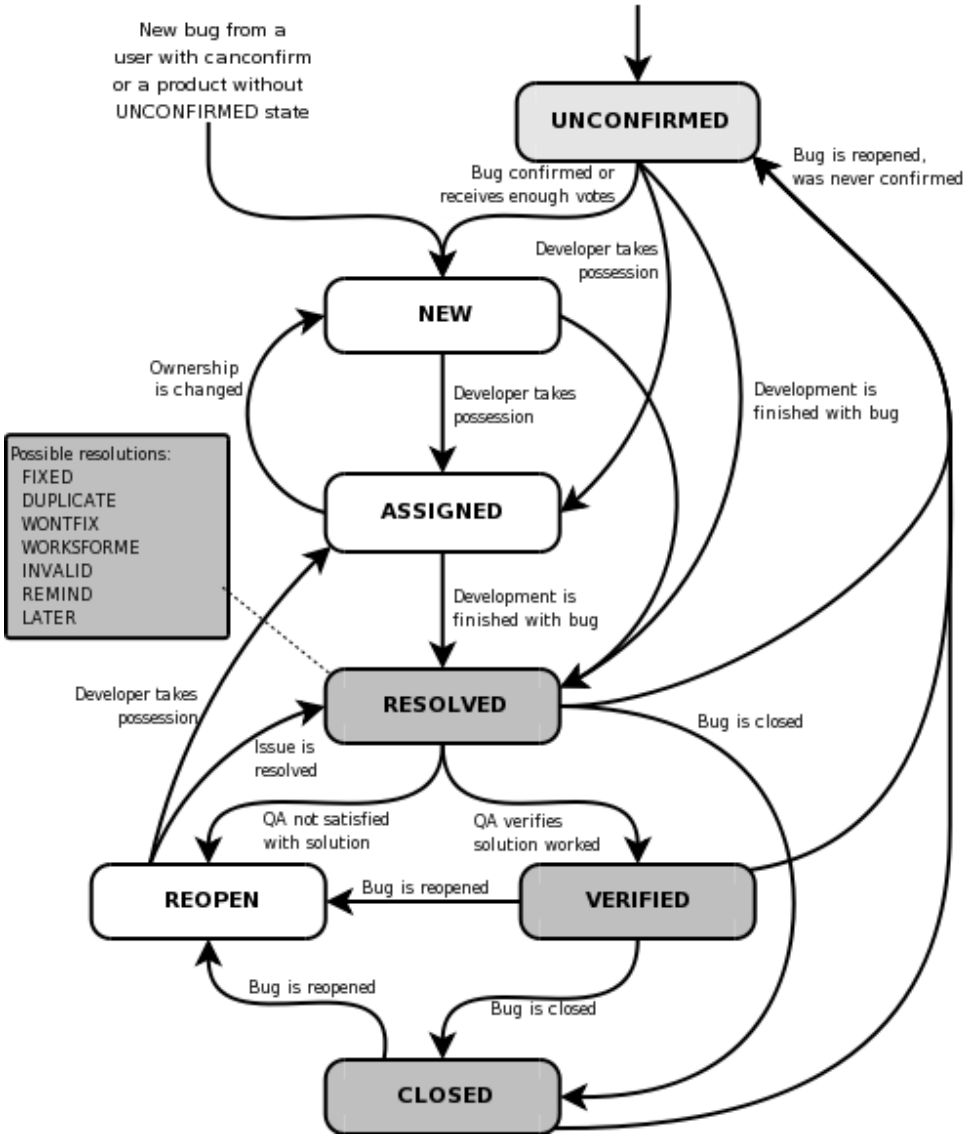
- **Reporter:** the person who found and reported the CR.
- **Created date:** field that marks the creation date of the CR.
- **Assignee:** the person who is working on analyzing or fixing the failure.
- **Updated/Modified date:** stores the modification dates referring to the CR, such as when a developer reviews and comments on something about the CR or when the CR is closed with some resolution.
- **Status/Resolution:** This field defines the current status of the CR, below, we describe more details about the different types of resolutions.
- **Affects Version/s:** field where the version in which the defect was found is informed.

Change Request Life Cycle

The CR life cycle, also known as workflow, is generally defined according to the way the company works, in this case, out of curiosity, we bring in Figure 4, a graphical representation

of the standard workflow used by the Bugzilla System. Depending on the status, the defect can be classified as an *Open Defect* or as a *Closed Defect*.

Figure 4 – Bugzilla Life-cycle of a Defect Report



Source: https://www.bugzilla.org

Open Defects are those that do not yet have a resolution associated with them, and these resolutions can be:

- **Unconfirmed:** This bug has recently been added to the database. Nobody has confirmed that this bug is valid. Users who have the *"canconfirm"* permission set may confirm this bug, changing its state to *New*. Or, it may be directly resolved and marked *Resolved*.
- **New:** This bug is valid and has recently been filed. Bugs in this state become *Assigned* when somebody is working on them, or become resolved and marked *Resolved*.

- **Assigned:** This bug is not yet resolved, but is assigned to the proper person who is working on the bug. From here, bugs can be given to another person and become *New*, or resolved and become *Resolved*.

Closed Defects, in turn, need to have a resolution associated with them, and these can be:

- **Resolved:** When they are closed as resolved, a resolution has already been assigned to this CR. However, it is still awaiting verification by a Quality Assurance (QA) team, which can, in fact, be verified and marked as *Verified* or *Reopened* to go through this CR's analysis process again. Once closed, this CR can be classified into one of the following statuses:
  - **Fixed:** A solution to the problem has been verified, integrated with the code, tested, and approved.
  - **Duplicated:** the open CR has already been reported. As a rule, they use the date of creation as a basis to define the CR that will be duplicated, usually, the oldest ones have higher priority.
  - **Wontfix:** the reported problem is actually a defect, but the company has defined that it will not fix it, often because it is a minor problem or a tough one to reproduce.
  - **Worksforme:** It may happen because what was described in the CR is not a problem, that is, it is in accordance with some product documentation, or because it was reproduced at the time the CR was opened, but at the time of analysis of this CR the problem was not reproducible anymore (i.e. it "works for me").
  - **Invalid:** The problem described is not a valid defect.
- **Verified:** When verified by the QA team and duly approved, in this case, marked as *Verified*, it means that the appropriate solution was taken by the development team, which is the final status of the CR.

2.4 CONCLUDING REMARKS

This chapter has briefly presented the concepts of Software Testing and its levels, types, and techniques, the Regression Testing techniques, focusing on the history-based prioritization technique. Finally, it described Manual Testing and its concepts, which in part are used by the TTC.

At TTC, the leading testing technique used is the black-box technique, mainly at the Component and System Testing levels. Regarding the types of tests, the Structural Test is the only type that is not performed in the TTC.

The Test Case Prioritization heuristics presented in this work follow the principles of history-based test case prioritization techniques.

3 RELATED WORK

This work is related to software regression testing and more specifically to test case prioritization techniques. Given that regression testing is a broad research topic, here we briefly cover the main regression testing techniques and give more emphasis to history-based prioritization. We also discuss a few related work in the context of manual testing.

3.1 REGRESSION TESTING

Regression testing is performed when changes are carried out in the software, and its objective is to provide confidence that the newly introduced changes do not break previously correct behavior of the existing, unchanged part of the software (YOO; HARMAN, 2012; ELBAUM; ROTHERMEL; PENIX, 2014; DO, 2016; KHATIBSYARBINI et al., 2018). As the software evolves, the test suite tends to grow, often making running the entire test suite very expensive (YOO; HARMAN, 2012; KHATIBSYARBINI et al., 2018). Given this limitation, it is often the case that practitioners need to rely on different regression testing optimization techniques to reduce the effort required for performing regression testing. The three main regression techniques are *test suite minimization* (or *test suite reduction*), *test case selection*, and *test case prioritization*.

Test suite reduction approaches try to find the smallest representative set of test cases, maintaining the original suite's failure detection capability but satisfying the tested software's requirements coverage. To define an ideal test suite reduction, it is recommended to compute the smallest reduced suite possible, but in order to preserve its failure detection capability similar to the original test suite, the ideal test suite reduction can only be achieved if the *reduced suite* contains a minimum number of test cases. *"Current test suite reduction strategies are grounded on various types of heuristics to produce approximate solutions within practical computational time"* (KHAN et al., 2018). Khan *et al.* (KHAN et al., 2018) bring as an objective, in their systematic review, the classification of test suite reduction approaches, evaluating the quality of these test suite reduction experiments based on defined quality criteria. With this, they were able to group the approaches into four main categories: greedy-based, clustering-based, search-based, and hybrid approaches, differing from each other by the algorithms used. Their systematic review results indicated that most test suite reduction-related researches are in the greedy-based category (69%), using one or more greedy algorithms to achieve faster

coverage and assuming that such coverage will also find more failures. The search-based approaches (20%) are gaining more attention due to the good results presented. In turn, hybrid approaches (8%) focus on combining the strengths of other TSR approaches to provide improved cost-effective solutions employing multiple algorithms. The authors observed that most test suite reduction approaches (80%) focus on solving the single-objective optimization problem. Khan *et al.* (KHAN et al., 2018) concluded that analyzing the quality of experiments related to test suite reduction approaches, the experiments reported do not adhere to the guidelines of well-designed software engineering experiments, which may present validity threats related to their results. Therefore, the works present in the literature analyzed so far by them did not have enough information about the guidelines for performing quality experiments in the field of test suite reduction. With that, their work can fill this research gap by providing a set of appropriate recommendations that researchers in the future can apply to conduct quality test suite reduction-related experiments.

Efficient regression testing is essential, even crucial, for organizations with their higher cost of software development. Among other tasks, this includes determining which test cases need to be re-executed to verify the side-effects of the modified software's behavior, which is the activity known as test case selection (or regression test selection). Regression test selection involves a trade-off between the cost of re-running the test cases and the risk of missing failures introduced by the side effects of software changes. This need for efficient regression testing strategies is thus becoming more and more critical. Engstrom, Runeson, and Skoglund (ENGSTRÖM; RUNESON; SKOGLUND, 2010) proposed in their work a goal to find a regression test selection technique that is empirically evaluated. During the study, they found that 28 different regression techniques for the regression test selection exist in the empirically evaluated literature. They also observed five major groups for these techniques that can be classified as *DejaVu-Based*, *Firewall-Based*, *Dependency-Based*, *Specification-Based*, and *Others*. DejaVu-Based is a group of techniques that Rothermel and Harrold proposed for procedural languages in 1993 (ROTHERMEL; HARROLD, 1993); Firewall-Based is a group of techniques where dependencies to modified software parts are isolated inside a firewall; Dependency-Based uses the dependency principle between more significant parts, such as classes or functions, leads to that all test cases using the changed part are re-executed even though the actual modified code may not be executed; Specification-Based is the technique that uses specifications or metadata of the software instead of the source code or executable code; and, Others are the techniques which share some similarities with either group, although not being directly derived

from one of them. The authors could notice a considerable variance regarding the uniqueness of the techniques identified in the papers analyzed. Some techniques may be considered novel at the time of their first presentation, while others may be regarded as only variants of already-existing techniques. As many variant techniques exist, there also exist many classifications of regression test techniques: *inclusiveness*, *precision*, *efficiency*, and *generality* (ROTHERMEL; HARROLD, 1996); *Minimization*, *Safe*, *Dataflow-Coverage-based*, *Ad hoc/Random*, or *Retest-All* (GRAVES et al., 2001); *techniques that operate at a higher granularity, e.g., method or class (called high-level) and techniques that use at a finer granularity, e.g., statements (called low-level)* (ORSO; SHI; HARROLD, 2004). Despite the groups of techniques and variants, Engstrom, Runeson, and Skoglund (ENGSTRÖM; RUNESON; SKOGLUND, 2010) highlight that the most commonly found property attributed to regression test selection techniques is whether these techniques are safe or unsafe. With a safe technique, the defects found with the entire test suite are also found with the test cases chosen by the regression test selection technique. This property can be used to classify all regression test selection techniques into safe or unsafe techniques. Two main categories of metrics were identified: cost reduction and failure detection effectiveness. One component of the cost for regression test selection is the analysis time required to select which test cases to rerun. However, in addition to saving costs, regression test selection techniques must detect as many failures as possible found by the original test suite. Engstrom, Runeson, and Skoglund (ENGSTRÖM; RUNESON; SKOGLUND, 2010) concluded that the regression test selection techniques might be classified according to: applicability on type of software and type of language; details regarding the method such as which input is required, which approaches are taken, and which level of granularity are changes considered; and properties such as classification in safe/unsafe or minimizing/not minimizing. And that the empirical evidence for differences between the techniques is not very strong and sometimes contradictory. However, they did not arrive at a basis for selecting a superior technique. Instead, they noted that the techniques must be adapted to specific situations.

Unlike the selection and reduction techniques, the test case prioritization technique values executing all cases that are present in the test suite but following a predefined order of execution of these test cases. Test case prioritization aims to order a set of test cases to achieve early optimization based on preferred properties. Giving the approach the ability to execute the most significant test cases first according to some measure to produce the desired result, such as revealing faults earlier by providing feedback to testers. This prioritization also helps find the optimal permutation of a series of test cases that can be executed accordingly, improving

testability in the software testing activity (KHATIBSYARBINI et al., 2018). One of the approaches of the work by Khatibsyarbini *et al.* (KHATIBSYARBINI et al., 2018) synthesizes an overview of the main approaches in prioritizing test cases, which were observed in 11 of the works evaluated by them, which are: *Search-based*, *Coverage-based*, *Fault-based*, *Requirement-based*, *History-based*, *Risk-based*, *Bayesian-Network-based*, *Cost-Effective-based*, and *Topic-Model-based*. The authors noted that test case prioritization continues to be recognized as an important element in regression testing among researchers because it has the ability to increase testing effectiveness in terms of failure detection rate, cost, and time. And each of the approaches has potential values, advantages, and limitations. Regarding the measurement metric, the APFD metric remains the primary metric used in TCP approaches. It is essential for any proposed approach to test case prioritization to perform metric measurements to assess their effectiveness. The evaluation metric is essential for measuring the effectiveness of any proposed TCP approach in prioritizing test cases and for comparing its effectiveness with other existing approaches. With their systematic literature review, Khatibsyarbini *et al.* (KHATIBSYARBINI et al., 2018) concluded that even though there are different types of approaches, the main objective of TCP in regression testing remains the same: to increase failure detection and that it has been recognized as an essential element in regression testing among researchers.

3.2 HISTORY-BASED PRIORITIZATION

The literature on test case prioritization is huge, and for a comprehensive overview, we refer the reader to several existing secondary studies on the topic (CATAL; MISHRA, 2013; HAO; ZHANG; MEI, 2016; KHATIBSYARBINI et al., 2018; BAJAJ; SANGWAN, 2019). The approaches closest to ours are those proposed by Kim and Porter (KIM; PORTER, 2002) and Haghighatkhah *et al.* (HAGHIGHATKHAH et al., 2018).

Kim and Porter (KIM; PORTER, 2002) proposed a history-based prioritization technique consisting of four steps: (1) first, they apply a test case selection technique to a suite T generating a suite T', (2) then they define a selection probability for each test of T', (3) choose the first case of T' using the probability value defined in step 2 and execute it[1], and (4) finally repeat step 3 until the end of the time available for carrying out the test execution. Their approach is based on statistical quality control (exponential weighted moving average)

[1] The test case to be chosen will depend on which approach is being used. In their work they have listed three different test history options that will provide different orderings.

and statistical prediction (exponential smoothing), defining the probability of selection of test cases with time-ordered observations extracted from previous runs and using a smoothing constant to weight individual historical observations, where a different test history will result in a different prioritization. In the evaluation conducted by the authors, the proposed history-based prioritization outperformed two random approaches used as control objects. Our approach differs from that proposed by Kim and Porter (KIM; PORTER, 2002) mainly because they consider only the last result produced by a test case, whereas in our approach, we consider the entire execution history of a test case since its creation for carrying out the prioritization.

Engstrom, Runeson, and Ljung (ENGSTRÖM; RUNESON; LJUNG, 2011) report a case study on the implementation of history-based regression testing to improve transparency and test efficiency at the function test level in a large software development organization. In their work, they highlight that there is a gap between research and practice of regression testing; even with the existence of several systematic approaches for both prioritization and selection of regression test cases proposed and evaluated in the literature, these approaches are not largely used in industrial setups, and this makes the application of regression testing highly dependent on people with experience of the product being tested. Another point that can be evident is that test managers add extra test cases to the scope to be on the safe side. Engstrom, Runeson, and Ljung (ENGSTRÖM; RUNESON; LJUNG, 2011) notice that a common approach for regression testing in the industry is to reuse a selection of test cases designed by testers, as this procedure is based on an individual's experience and judgment, it is not transparent enough to enable a consistent assessment of the extent and quality of regression testing, and, there is no direct evaluation of its efficiency. During their study, they observed that the share of industrial evaluations of regression test techniques is low. This is not unique to the regression testing field but rather general. Usually, the studies are conducted on the same set of artifacts which is good from a benchmarking perspective but limits the external validity since the programs are relatively small. Even with industrial artifacts, offline studies tend to reduce the complexity of the real task of test case selection in the industry. So their report is a case study evaluating history-based regression testing in an industrial setting to improve the transparency of regression testing selection and prioritization since less than a third of the studies comprise industry-scale contexts. Their context was focused only on black-box approaches since the testers do not have full access to the code. Their proposed improvements are based on the ideas from Kim and Porter (KIM; PORTER, 2002) and are implemented as suggested by Fazlilzadeh *et al.* (FAZLALIZADEH et al., 2009). Their objective was to improve

regression testing at the function test level by adapting and implementing history-based regression testing into the current context. In this case, improvements referred to increased test scope selection procedure transparency and increased or maintained test effectiveness. The exploratory step further supported the hypothesis that history-based regression test selection could improve the current situation as a starting point for this case study. Engstrom, Runeson, and Ljung (ENGSTRÖM; RUNESON; LJUNG, 2011) conclude that their quantitative evaluation showed that history-based prioritization of an already selected test suite improves the ability to detect faults early, which is good if a test session is prematurely interrupted. More important is the efficiency of a selection based on total prioritization. At the same time, a worst-case interpretation of available data shows a slight decrease in efficiency.

The work presented by Haghighatkhah *et al.* (HAGHIGHATKHAH et al., 2018) focuses on two approaches to test case prioritization, which are the history-based prioritization heuristic and the diversity-based prioritization heuristic. The diversity-based prioritization assumes that similar test cases that exercise the same part of the system are likely to detect the same faults. Therefore, a diverse set of test cases must be executed in order to detect a greater number of failures. Diversity-based TCP requires minimal information since the only required information is already encoded in the test suite. The authors reinforce that to increase the probability of catching failures, a potential strategy is to assign a higher priority to test cases that are more different from those already prioritized. In (HAGHIGHATKHAH et al., 2018) the authors propose the combined use of the diversity-based prioritization technique together with history-based test prioritization in a continuous integration environment. They investigated history-based diversity using three different similarity metrics: Manhattan, Normalized Compression Distance, and Normalized Compression Distance Multiset, and during these investigations, they were evaluated using different history interval sizes. Diversity-based TCP can be implemented using different methods and at different levels, for example, source code, test code, method calls, topic templates extracted from test cases, or English texts from manual test cases. To achieve diversity-based TCP, the dissimilarity between test cases must be calculated using a specific method. Haghighatkhah *et al.* (HAGHIGHATKHAH et al., 2018) calculated the cumulative priority for each test using their previous failures over the last N builds. The highest weight corresponds to the failure exposed in a previous build ($current - 1$), and the failure in every preceding build is weighted lower than the failure in its successive build. They aggregated tests into clusters based on their weight and sorted these clusters in descending order, then they compared the effectiveness of history-based prioritization with random permutation and

with each other using different interval sizes. Using a history-based prioritization approach, test cases without historical failure are grouped in a single cluster and remain in their original order. There was also the possibility that the investigated projects could have already used TCP techniques. To avoid the impact of this and to create a fair comparison with random ordering, they simply randomized the intra-cluster tests, this technique is called history-based random (HBR) in their study. In the technique presented in (HAGHIGHATKHAH et al., 2018), tests that do not have enough information regarding previous failures cannot be effectively prioritized. These tests are grouped into a simple cluster that is prioritized based on previous failure knowledge. This includes both newly added tests and those which have not revealed any failure in previous builds. Furthermore, within each cluster, there might be several tests with the same weight. To break the tie, the intra-cluster tests can be randomized, or other TCP techniques can be used in combination with history-based test prioritization. The results of the experiments conducted by Haghighatkhah *et al.* (HAGHIGHATKHAH et al., 2018) suggest that historical knowledge of failures appears to have strong predictive power in continuous integration environments and can be used to prioritize test cases for execution effectively; history-based prioritization does not necessarily require a large amount of historical data and its effectiveness improves to some extent with a larger historical range; Diversity-based TCP can be used effectively during the early stages of software development when historical data is not yet available or is scarce, and also combined with history-based test prioritization to improve its effectiveness; And finally, they found out that history-based diversity using normalized compression distance multiset is superior in terms of effectiveness, but this result comes with relatively high overhead in terms of method execution time. Given that it was possible to apply the approach by Haghighatkhah *et al.* at the context where we conducted our study this is the state-of-art approach that we considered as competitor in our evaluation (Chapter 5).

Fazlalizadeh *et al.* (FAZLALIZADEH et al., 2009) address the prioritization of test cases based on past execution history, considering time and resource constraints. While the approach by Kim and Porter (KIM; PORTER, 2002) considers only the last test run to calculate the test selection priority, Fazlalizadeh *et al.* (FAZLALIZADEH et al., 2009) propose to consider the time and resource constraints of the environment and incorporate three factors in their approach: historical failure detection effectiveness, execution history of each test case, and the last priority assigned to the test case. The detection effectiveness is computed as the ratio between the number of times a test failed over the total number of times it has been executed so far. Another factor that they also take into account is the time period in which the test case is not

executed because they seek to guarantee that each test in the test suite should be executed at some point. And lastly, they define a code coverage percentage for each test case based on various control-flow and data-flow criteria. Our approach is similar to the one proposed by Fazlalizadeh *et al.* (FAZLALIZADEH et al., 2009) in the sense that both approaches consider the entire execution history. The main difference, however, is that our approach does not weigh the test results depending on how long ago the tests were performed as done in (FAZLALIZADEH et al., 2009).

Najafi, Shang, and Rigby (NAJAFI; SHANG; RIGBY, 2019) adopted approaches based on historical test failure frequency, test failure association, and the costs associated with the testing process. For the test selection phase, the authors adopted approaches based on prior test failure frequency, association, and the costs of the testing process. Frequency: for the authors, intuitively, tests that previously failed frequently are more likely to fail again later, so the frequency of past test failures can be used as an indicator for suggesting test selection opportunities. Association: the authors noticed in previous studies that there are many co-failures in test executions; hence, associations can be effectively leveraged to improve test effectiveness. Cost: Test execution comes with a cost, especially for the complex testing infrastructure used by the company related to their study; with such high testing costs, they notice that tests can be skipped if skipping them is more cost-effective than executing them. For the test prioritization phase, the authors reported that their approach relies only on the test execution history, such as the number of true-positive failures and test execution duration, to calculate the priority values. As the information related to coverage is not available in the company where the study was carried out, the authors do not use test coverage information; similar to test selection approaches, they learn about all the test execution results until one day before every test. Related to the metrics used to validate their work, Najafi, Shang, and Rigby (NAJAFI; SHANG; RIGBY, 2019) use three metrics to evaluate the approaches for test selection (total test execution time reduction, number of slip-through test failures, and total cost reduction) and two other to assess the test prioritization (reaching the first test failure, and reaching all test failures as early as possible). Regarding this, they concluded that test prioritization by simply considering the past effectiveness of the tests could significantly help reduce the time to reach the first failure and provide a close-to-optimal order for running the tests. Their experiment reveals that industry experts are reluctant to remove tests from the test flow unless they find it necessary. There is always a relatively conservative approach for test executions by the preference for increasing the level of quality assurance of the software with the price of spending more

time and resources. To minimize this, they followed the approach instead of only providing historical evidence; they aimed to explain why those tests could be removed. For example, they can demonstrate that a specific *'Test A'* can be removed since the exercised *'feature'* is obsolete, instead of only showing that *'Test A'* has never detected a bug in the system. With their work, Najafi, Shang, and Rigby (NAJAFI; SHANG; RIGBY, 2019) observed that the results show the standalone test prioritization approach significantly outperforms all of their 1. selection approaches, 2. the combination of their prioritization and selection approaches, and also 3. the random order. With that, they conclude that test prioritization is the most effective and the least invasive approach for saving costs in testing processes.

The works presented by (ENGSTRÖM; RUNESON; LJUNG, 2011) and (NAJAFI; SHANG; RIGBY, 2019) are two works very closely related to ours; the context of their work is in a real industrial environment, like ours. In both cases, however, the work's primary focus was on the selection of test cases rather than the prioritization of the test suite. This differs from our work as we focus on the test case prioritization phase. As their work is related to a real industrial environment, like ours, they also chose not to expose much information about the company in question, which makes it somewhat challenging to reproduce both experiments.

3.3 MANUAL TESTING PROCESS

Although the literature is vast in works that address the subject of regression testing and its techniques, few works can be found that report the application of these techniques in the context of manual testing.

Hass *et al.* (HAAS et al., 2021), in their work on how manual testing processes can be optimized, explain a little about the application of manual testing in some real-life contexts. They list some difficulties and challenges that demonstrate why the use of manual tests is still a common practice and commonly used by the industry. According to their observations, manual testing is often used because it is closer to reality, it has a more specific context, and it is up-to-date, not to mention that it is even more sustainable when complexity is high, and requirements are blurred (based on the answers obtained by the questionnaire applied by the authors). In addition, some industries, such as those in the medical field, determine the use of manual tests. In addition to these challenges that prevent test automation, some other obstacles are mentioned, such as lack of time, lack of budget, limited know-how, limited technology, high change frequency, etc. Another aspect observed by the authors was the way

test cases are selected and assigned (allocated) to testers during execution; they observed that this method differs a lot from one company to another. Regarding the companies' approach for test case selection and prioritization, the selections were based on: code changes, tester experience, feature criticality, requirements, time constraints, or test failure history. Regarding prioritization, this number was very low, approximately 8% of the participants mentioned the use of prioritization techniques when they are selecting the tests, and the techniques used by them were based on: experience, licensing or hazard relevance. When assigning test cases to testers, the following were considered: tester experience and areas of responsibility. Hass *et al.* (HAAS et al., 2021) concluded that manual testing is still widely used in the industry, despite the high cost of human effort involved in this process. And that according to the research they've done on these companies that focus on manual testing, they don't have any intention of fully automating their tests in the near future. As reported by Hass *et al.* (HAAS et al., 2021), the application of manual testing is still a common practice precisely because it is the closest scenario to reality; it is in this context that we can achieve a behavior very close to the behavior of the end-user of the product. The authors provide some suggestions for optimizing manual testing processes and show that the application of prioritization techniques in the manual context is still shallow. In our work, we propose and evaluate history-based prioritization heuristics precisely for the context of manual testing.

Another work related to the execution of manual tests is the one proposed by (LIMA, 2009). The work proposes a technique for prioritizing test cases based on data reuse, focused on the permutation that generates all possible sequences of a set of test cases to reduce the time spent configuring these executions and to improve test team productivity. The main contribution of this work is the evaluation of four test case prioritization techniques that use the concept of data reuse to generate test sequences. The author focuses on a permutation technique that guarantees the computation of the best sequences (there may be more than one) of test suites with a small number of tests to be used as an exploratory algorithm. With this study, (LIMA, 2009) provides empirical evidence that the permutation technique can generate better sequences than those generated manually. The work shows that reliable results can be provided and that these permutations present good results when compared with manual prioritization heuristics. However, these permutations are unfeasible for large test suites. Even the other three techniques based on classical artificial intelligence techniques (greed algorithm, simulated annealing, and genetic algorithm) do not guarantee a better result. The context where the study of (LIMA, 2009) was conducted is very similar to ours. We could

not use this work as a competitor because the proposed technique is aimed at small suites with an approximate number of 15 test cases per test suite.

3.4 CONCLUDING REMARKS

In this chapter, we presented some of the works in the literature that address the main topics related to this dissertation, among them studies related to the main regression testing techniques, the history-based prioritization approach, and the manual execution of test cases.

4 OUR APPROACH

In this chapter, we discuss our approach. We introduce two history-based prioritization heuristics, and we discuss the rationale behind their proposal. Before introducing the proposed heuristics, we provide additional details related to the industrial context where this study was carried out.

4.1 CONTEXTUALIZATION

The company where our study was conducted provides testing services to a smartphone manufacturer. The products developed by TTC are designed to be marketed worldwide and each different region has its own demands, meaning that the software needs to be customized and tested accordingly for every target region.

The tests performed by TTC aim at guaranteeing the quality of the software that ships with the product as well as the quality of any new versions released for the purpose of upgrading the software of existing products. The majority of the test cases are executed *manually*.

The products tested at TTC can be classified into different categories depending mainly on their hardware specifications. For example, some products could be referred to as *entry-level*, whereas others could be referred to as *high-tier* — generally more powerful products in terms of hardware. In this work, we refer to these categories as *families* to facilitate understanding. Products from the same family share many features, which means that the same test suite (or part of it) can be used to assess the quality of many products from the same family. These products are tested at each life cycle of a new Operational System (OS) version, which in the case of this company's products uses Android[1], a linux-based OS that operates on smartphones, laptops, and tablets. It is developed by the Open Handset Alliance[2], an alliance between several companies, including Google[3].

The frequency with which each product is tested depends on many aspects, including the product's family. In general, high-tier products go through more testing cycles than those from the entry-level family.

The test suites are organized based on the target region and on the application to be tested.

[1] https://source.android.com/
[2] http://www.openhandsetalliance.com/
[3] https://about.google/

When a new test cycle is triggered (for example, whenever a new operating system version is released), the Test Manager (TM) needs to define which test cases should be performed. For carrying out this task, the TM takes into account many aspects, including the product schedule, the number of test cycles running in parallel, and the number of testers available for running the selected test cases, among others.

In addition to these families sharing similar characteristics between their products, they also share the test cases used to test all these products, whether for products in the same family or even in different families.

These test cases are organized according to a specific region of the platform's code being tested or according to the base application. Within the regression team, an area that is being used as a reference for the development of this project, the test cases are divided into eleven different categories, which are: *Core, AOSP, GMS, Messages, Dialer, Google Pay, Android Auto, Experiences, Dual SIM, Carrier, and Data Migration*, and each area has its set of tests that serve as a basis for the creation of test suites according to the product that will be tested. The approximate numbers of these tests are listed in Table 1.

Table 1 – Number of Test Cases by Area

Area	**TC#**
Core	710
AOSP	140
GMS	130
Google Pay	25
Android Auto	30
Message	45
Dialer	90
Experiences	350
Carrier	150
Dual SIM	200
Data Migration	170

With each new product that is tested, the Product Owner (PO), together with the Subject Matter Expert (SME) (or we can refer to them as Test Managers), defines the test plan that will be executed for that product and the number of test cases that will compose each test suite. At this moment, a selection of test cases is made that will contemplate the coverage of this product. Once all the plans have been created and the test strategy is organized, the test suites are passed on to the testers to execute all test cases in a time window that is defined

according to the product schedule, the number of products that are running in parallel, and the number of testers available to run all available test suites for the products.

The frequency with which each of these products is tested, taking into account the execution of all the categories mentioned above, will depend on the family in which this product belongs. As a rule, the products that are from the entry-level and intermediate families pass only for two lifecycles of test executions: once on the version in which the product was released, and a second time when that product receives an OS upgrade. High-tier family products usually go through three life cycles of test executions: one when it is released, plus two others OS upgrades for this product.

At TTC the TM can take advantage of an existing tool, developed in-house, that will help with the test case selection task. Based on the information on recent changes performed in the software, the tool can automatically select a set of suitable test cases that will cover the intended parts of the software. The tool can also automatically export the selected test suite to the test management tool where the test results will be reported by the testers.

4.2 PROBLEM

While there is automated support for the test case selection task, no support exists for the test case prioritization phase, i.e., after the test cases are exported from the test management tool (or when the TM create the test suite manually) they are displayed ordered by the test case IDs. Without a clear ordering for the test cases, it is left for the tester to decide in which order the test cases should be executed — and as we will see in the results of our empirical evaluation in Section 5.8, the tester choice is far from optimal.

Based on what has been presented so far, we can see that we talked about defining: how the test cases need to be executed, how to group the test cases, or how to order them by following some specific pattern.

In this case, TTC tries to guarantee that exactly all the test cases defined in the different test strategies are executed.

In order to optimize execution, technical leaders always advise testers to group the test cases within the test suites according to domain similarities of the test suites being tested so that this grouping somehow optimizes the execution.

4.3 SOLUTION

To overcome this problem we suggest that TTC could take advantage of its large database of historical test results and adopt a fully-automated history-based test case prioritization approach. To support our intuition, in the next section, and on subsequent ones, we propose two different history-based TCP heuristics and we empirically evaluate their performance in the context of TTC.

As no test case prioritization technique has been used so far, the main objective of this work is to propose the use of these prioritization techniques that aims to optimize how these cases are ordered to be tested, as well as the possibility of an improvement in the failure detection rate earlier in the product testing phase.

As there are different test areas and different types of test suites (according to its area), for this study in question, the trial will be based on one of the test areas that have a more significant number of manual test cases, as well as the focus will be on the biggest test suite within this area, which is the *Core Regression Test Suite*, as we can see on Table 1.

For this, two different heuristics will be presented and compared with some *controls* heuristics (*optimal, real, default, and random*), as well as two other heuristics related to History-based prioritization presented in the literature as the *state-of-the-art*, these heuristics will be shown in Section 5.3.3.

4.4 HEURISTICS

To overcome the limitations of TTC described in Section 4.2 we propose the adoption of simple, yet effective and efficient, history-based prioritization heuristics by leveraging the large database of historical test results available at the company.

Simply put, our history-based prioritization heuristic orders the test cases in descending order of their past *failure-frequency* and the cases of ties are solved by considering the *test-age* — failure-frequency is computed by dividing the number of times a test failed in the past with respect to its total number of executions, whereas test-age counts the number of days since the test case was created or updated. Priority is given to failure-frequency because it has been shown that a test that failed in the past tends to fail again (KIM; PORTER, 2002). On the other hand, priority is given to newly-added or modified test cases because such tests are usually added (or updated) to cover new features or updated parts of the SUT and, intuitively, those

could be error-prone areas.

Algorithm 1 provides a pseudo-code with a more detailed description of our history-based prioritization strategy. First, we collect the list TS of test cases available in the test suite created by the test manager (line 2). Then, we collect the execution history for each test case in the list TS (line 4). The function $getFailureFrequency()$ at line 5 processes the execution history to compute the failure-frequency for each test case, while function $getTestAge()$ at line 6 gets the date at which each test case was created (or updated). The prioritization itself happens at function $orderTestCases()$ (line 7) and the test cases are added to the test suite PTS in descending order of their past failure-frequency with the cases of ties being solved by considering the test-age in ascending order. Finally, the algorithm outputs the prioritized test suite PTS (line 8). The function $getHistory()$ (lines 9 to 14) collects the execution history for all test cases in the list TS. It receives, as input, the test suite and the heuristic used (the available heuristics are detailed next).

```
   input  : Test suite info ts, and heuristic
   output: Prioritized test suite PTS
 1 PTS ← emptyList();
 2 TS ← getTestCaseIDs(ts);
 3 function prioritizeTestSuite()
 4 |   history ← getHistory(TS, heuristic);
 5 |   failure_frequency ← getFailureFrequency(TS, history);
 6 |   test_age ← getTestAge(TS);
 7 |   PTS ← orderTestCases(TS, failure_frequency, test_age);
 8 |   return PTS;
 9 function getHistory(TS,heuristic)
10 |   if heuristic == 'family-dependent' then
11 |   |   history ← getExecutionResults(TS, filter = true)
12 |   else if heuristic == 'family-independent' then
13 |   |   history ← getExecutionResults(TS, filter = false)
14 |   return history;
```

Algoritmo 1: History-based prioritization

For carrying out our experiments (detailed in Chapter 5) we explored two variations of our history-based prioritization strategy that control which entries of the execution history are considered for the prioritization:

- Because each product at TTC belongs to a family of products that share many characteristics, including specifications and core software components, one heuristic filters

the execution history of a test case to consider only the results that were reported for products that belong to the same family of the target product. We refer to this heuristic as **family-dependent**.

- Alternatively, we consider the entire execution history of a test case, regardless of the product or software version for which the test case was executed. We refer to this heuristic as **family-independent**.

Note that Algorithm 1 is the same for both heuristics and the only place where it differentiates between family-dependent and family-independent is in the function *getHistory()* (lines 9 to 14): for family-dependent the history of executions is filtered (line 11), whereas for family-independent the entire history of executions is considered (line 13).

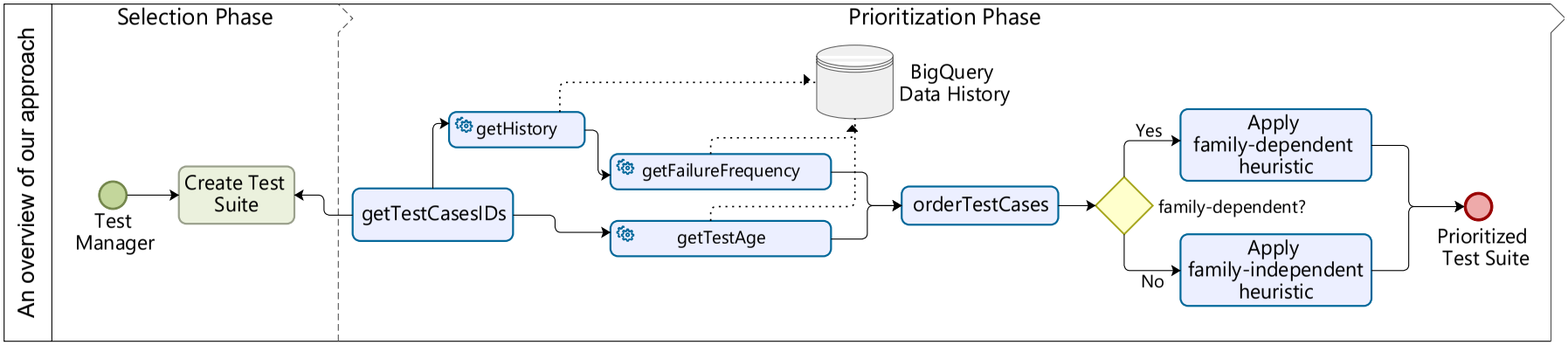
While Algorithm 1 presents a micro perspective of the proposed approach, Figure 5 presents the macro perspective: it depicts the complete flow of our approach in the context of TTC. During the *selection phase* the test manager selects which test cases should be added to the test suite. During the *prioritization phase* our approach collects all the required information for the prioritization (execution history, failure frequency, and test age). The execution history is stored in a serverless multicloud data storage module, the BigQuery[4], which stores data that is designed precisely for querying extensive data information. Finally, the test cases are ordered according to the chosen heuristic (family-dependent or family-independent) and a prioritized test suite is produced.

4.5 CONCLUDING REMARKS

In this chapter, we presented the context and motivation for proposing our approach. We introduced two history-based prioritization heuristics and we discussed the rationale behind their proposal. The proposed heuristics were discussed both at a pseudo-code level but also with a high-level overview of the industrial context where we conducted our study.

[4] https://cloud.google.com/bigquery

Figure 5 – History-based Prioritization in the Industrial Context where we Conducted our Study



Source: Created by the author

5 EVALUATION

This chapter presents the experimental study that evaluates the effectiveness and efficiency of the proposed heuristics. We start by providing the goal definitions, followed by the study planning, preparation and analysis. We then discuss the threats to the validity of our study. Finally, we address the execution and discuss the results of our experiment.

5.1 STUDY PLAN

To make sure that our results are reliable we follow the guidelines of Juristo and Moreno (JURISTO; MORENO, 2013), and Wohlin *et al.* (WOHLIN et al., 2012) for conducting experimentation in software engineering. We also provide all the information needed to allow possible replications of our study, including the study plan and goals, hypotheses, treatments, control objects, experimental objects, variables, as well as the planning, preparation, execution, analysis of the results, and threats to validity.

5.2 GOAL DEFINITION

Following the guidelines by Basili (BASILI, 1994) we defined the following goals, questions and metrics for our study.

5.2.1 Global Goal

Distinguish our proposed history-based prioritization approaches from competitor approaches.

5.2.2 Measurement Goal

Considering our proposed history-based prioritization approaches, we plan to characterize their difference with respect to competitor approaches, concerning:

- Impact on effectiveness: measuring how effective our proposed approaches are with respect to our competitors;

- Impact on efficiency: measuring how efficient our proposed approaches are with respect to our competitors;

5.2.3 Study Goal

- **Analyze** the performance of the designed history-based prioritization approaches.
- **For the purpose of** comparison.
- **With respect to** alternative prioritization strategies.
- **From the viewpoint of** software testers and test managers.
- **In the context of** manual testing in a real industrial setting.

5.2.4 Questions

For assessing our goals we defined the following questions:

RQ1 **[Effectiveness]:** How effective are the proposed history-based approaches when compared with alternative prioritization strategies?

RQ2 **[Efficiency]:** How efficient are the proposed history-based approaches when compared with a state-of-the-art approach?

5.2.5 Metrics

The assessment of RQ1 and RQ2 are performed by metrics M_1 and M_2, respectively.

M_1 **[APFD]:** Measures the Average Percentage of Faults Detected (APFD) (ROTHERMEL et al., 1999), the *de facto* metric used for evaluating TCP approaches. This is confirmed by recent secondary studies that surveyed the topic of TCP (CATAL; MISHRA, 2013; HAO; ZHANG; MEI, 2016; KHATIBSYARBINI et al., 2018). The APFD can be computed according to Equation 2.1, where m is the number of faults found in a test suite, n represents the total number of test cases, and TF_j represents the first test case in the prioritized test suite that detects the fault j. The APFD value can vary from 0 to 1, and the higher the value, the faster faults are detected by a test suite.

M_2 **[Time]:** Measures the *total time* required for prioritizing test suites using the investigated history-based prioritization approaches. Total time considers both preparation time (e.g., collecting the historical data) and prioritization time (the time required for ordering the test suite).

5.3 PLANNING

Based on the goal, questions and metrics defined for our study, here we provide additional information regarding the planning of our experiments, including our hypotheses, treatments, dependent and independent variables.

5.3.1 Hypotheses Definition

To address RQ1, our null hypothesis states that no statistical difference can be observed in the speed at which faults are revealed when our proposed approaches are compared with alternative strategies. The alternative hypothesis, the one to be accepted in case the null hypothesis is rejected, states that differences can be observed:

Null Hypothesis ($H0_1$): The median APFD values achieved by our prioritization approaches and the alternative prioritization strategies will not differ.

$$H0_1 : APFD_{ours} \simeq APFD_{competitors}$$

Alternative Hypothesis ($H1_1$): The median APFD values achieved by our proposed approaches are different from those achieved by the alternative strategies.

$$H1_1 : APFD_{ours} \neq APFD_{competitors}$$

To address RQ2, our null hypothesis states that no statistical difference can be observed in the total prioritization time required by our prioritization approaches and the alternative prioritization strategies. The alternative hypothesis states that differences can be observed:

Null Hypothesis ($H0_2$): The total prioritization time required by our prioritization approaches and the alternative prioritization strategies will not differ.

$$H0_2 : TIME_{ours} \simeq TIME_{competitors}$$

Alternative Hypothesis ($H1_2$): The total prioritization time required by our proposed approaches are different from those achieved by the alternative strategies.

$$H1_2 : TIME_{our} \neq TIME_{competitors}$$

5.3.2 Treatment

The treatment used in this study is the prioritization of test suites based on the execution history of test cases. We can say that this is a one-factor experimental study since we use only one treatment: the application of the prioritization heuristics. The absence of treatment is the use of alternative prioritization strategies (also referred to as competitors in our study).

5.3.3 Control Object

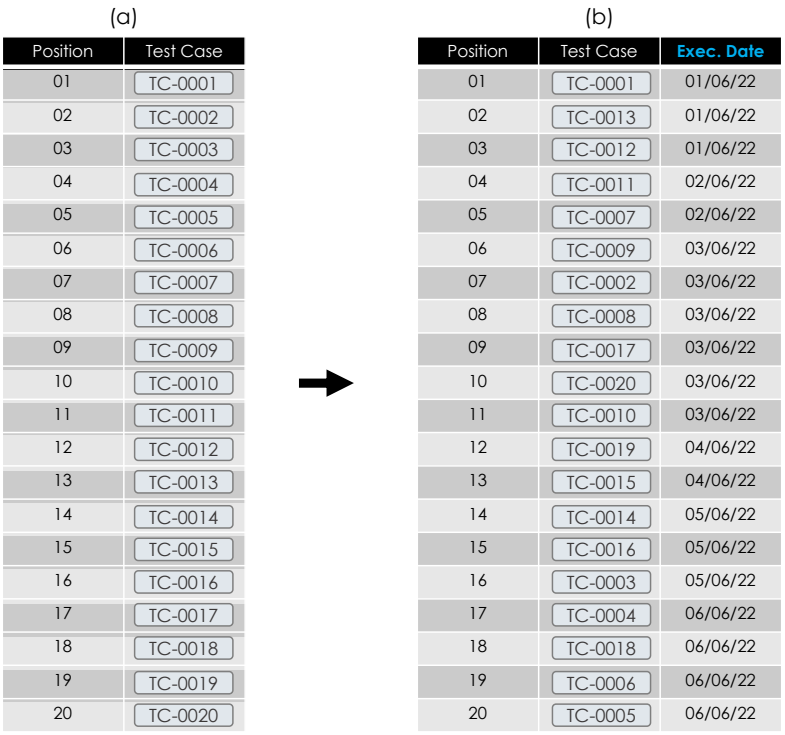
To identify the impacts of using the history-based prioritization approaches, the control object is the use of the data from previous executions where the test cases were executed using the testers' ad-hoc prioritization.

This includes the alternative orderings we use for comparison (competitors):

- Real prioritization (*RealOrd*): this is the real ordering followed by the testers when executing the test cases available in the test suite. We collect this information from the database of historical test results available at TTC. To illustrate, the test cases available at the test suite in Figure 6 (a) are ordered following their real *Execution Date* displayed in Figure 6 (b).
- Default prioritization (*NewOld*): the test suite is ordered by the test management tool used in the context where our study was carried out. To illustrate, in the default ordering the test cases in Figure 7 (a) are ordered based on their creation date, with the newer test cases[1] being executed first, Figure 7 (b).

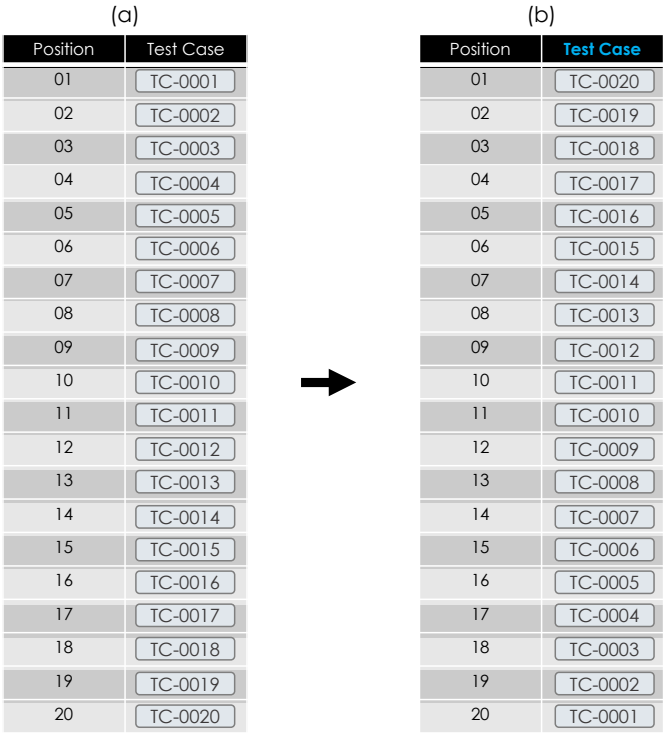
[1] The new test cases are the ones with the higher TC-ID value.

Figure 6 – Real Prioritization Ordering



Source: Created by the author

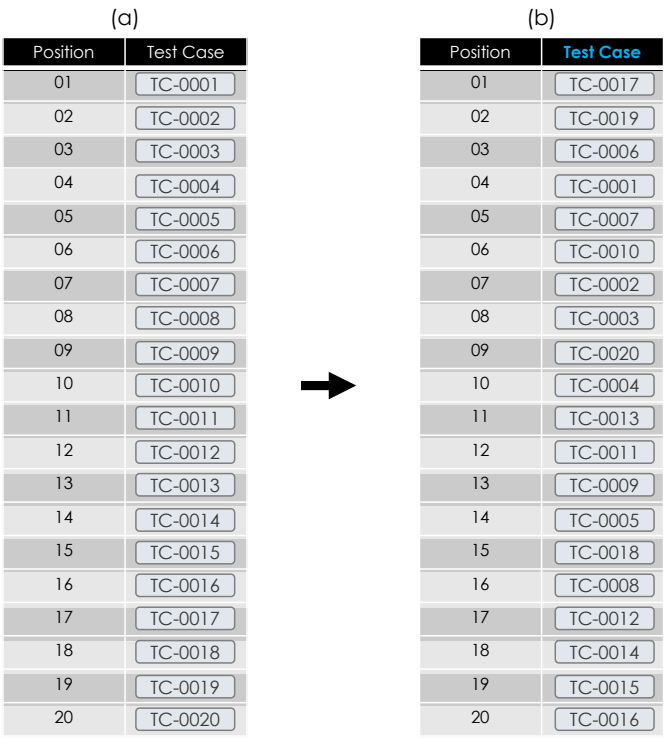
Figure 7 – Default Prioritization Ordering



Source: Created by the author

- Random prioritization (*Random*): the test suite (Figure 8 (a)) is ordered randomly. We derive such an ordering by shuffling the tests available in the test suite (Figure 8 (b)).

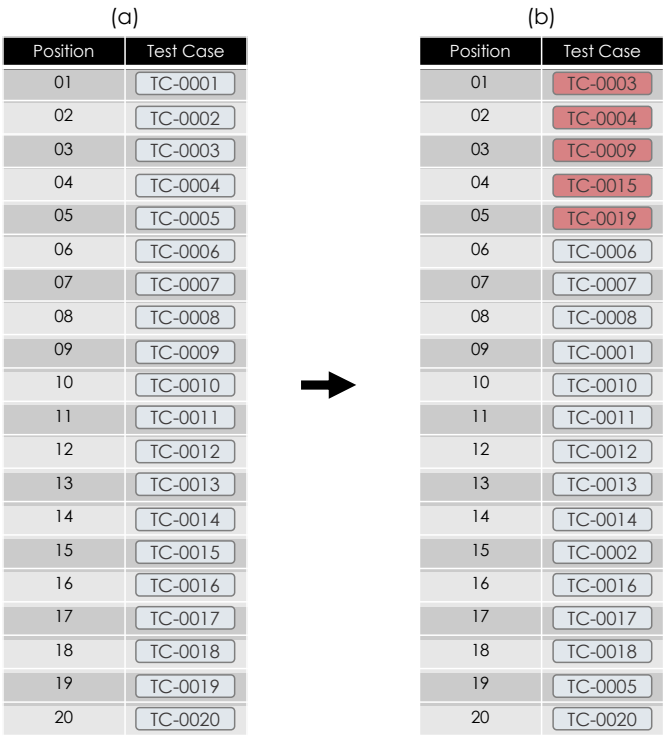
Figure 8 – Random Prioritization Ordering



Source: Created by the author

- Optimal prioritization (*Optimal*): the test suite is ordered in such a way that the maximum possible APFD is achieved. We derive such a test suite by first putting all the fault-revealing test cases (highlighted in red), followed by all the non-fault-revealing tests. This ordering is illustrated in Figure 9.
- History-Based Diversity (*HBD*): is a state-of-the-art competitor. The priority of each test is calculated taking into account the previous failures of the last n builds of the software, where the highest weight is assigned to the failure that was found in the immediately previous build with respect to the one being tested. In this heuristic the faults are weighted by their distance n (Wn), Figure 10 (a). Then, the test cases are grouped into clusters considering their Wn value, and test cases that do not have historical failures are grouped into a single cluster, Figure 10 (b). Lastly, the intra-cluster tests are ordered based on their distance (dissimilarity) to the set of already-prioritized tests (represented here by the color spectrum), and the test cases that do not have historical failures remain in their original order, Figure 10 (c).

Figure 9 – Optimal Prioritization Ordering



(a)

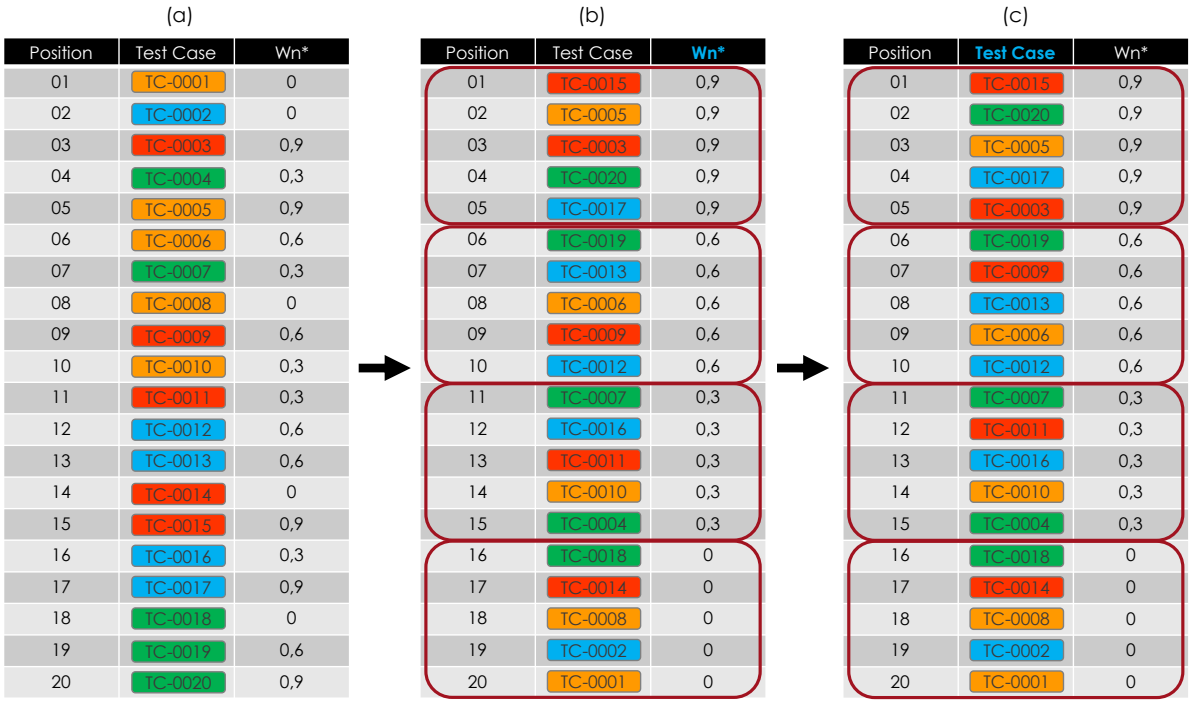
Position	Test Case
01	TC-0001
02	TC-0002
03	TC-0003
04	TC-0004
05	TC-0005
06	TC-0006
07	TC-0007
08	TC-0008
09	TC-0009
10	TC-0010
11	TC-0011
12	TC-0012
13	TC-0013
14	TC-0014
15	TC-0015
16	TC-0016
17	TC-0017
18	TC-0018
19	TC-0019
20	TC-0020

(b)

Position	Test Case
01	TC-0003
02	TC-0004
03	TC-0009
04	TC-0015
05	TC-0019
06	TC-0006
07	TC-0007
08	TC-0008
09	TC-0001
10	TC-0010
11	TC-0011
12	TC-0012
13	TC-0013
14	TC-0014
15	TC-0002
16	TC-0016
17	TC-0017
18	TC-0018
19	TC-0005
20	TC-0020

Source: Created by the author

Figure 10 – History-Based Diversity Ordering



(a)

Position	Test Case	Wn*
01	TC-0001	0
02	TC-0002	0
03	TC-0003	0,9
04	TC-0004	0,3
05	TC-0005	0,9
06	TC-0006	0,6
07	TC-0007	0,3
08	TC-0008	0
09	TC-0009	0,6
10	TC-0010	0,3
11	TC-0011	0,3
12	TC-0012	0,6
13	TC-0013	0,6
14	TC-0014	0
15	TC-0015	0,9
16	TC-0016	0,3
17	TC-0017	0,9
18	TC-0018	0
19	TC-0019	0,6
20	TC-0020	0,9

(b)

Position	Test Case	Wn*
01	TC-0015	0,9
02	TC-0005	0,9
03	TC-0003	0,9
04	TC-0020	0,9
05	TC-0017	0,9
06	TC-0019	0,6
07	TC-0013	0,6
08	TC-0006	0,6
09	TC-0009	0,6
10	TC-0012	0,6
11	TC-0007	0,3
12	TC-0016	0,3
13	TC-0011	0,3
14	TC-0010	0,3
15	TC-0004	0,3
16	TC-0018	0
17	TC-0014	0
18	TC-0008	0
19	TC-0002	0
20	TC-0001	0

(c)

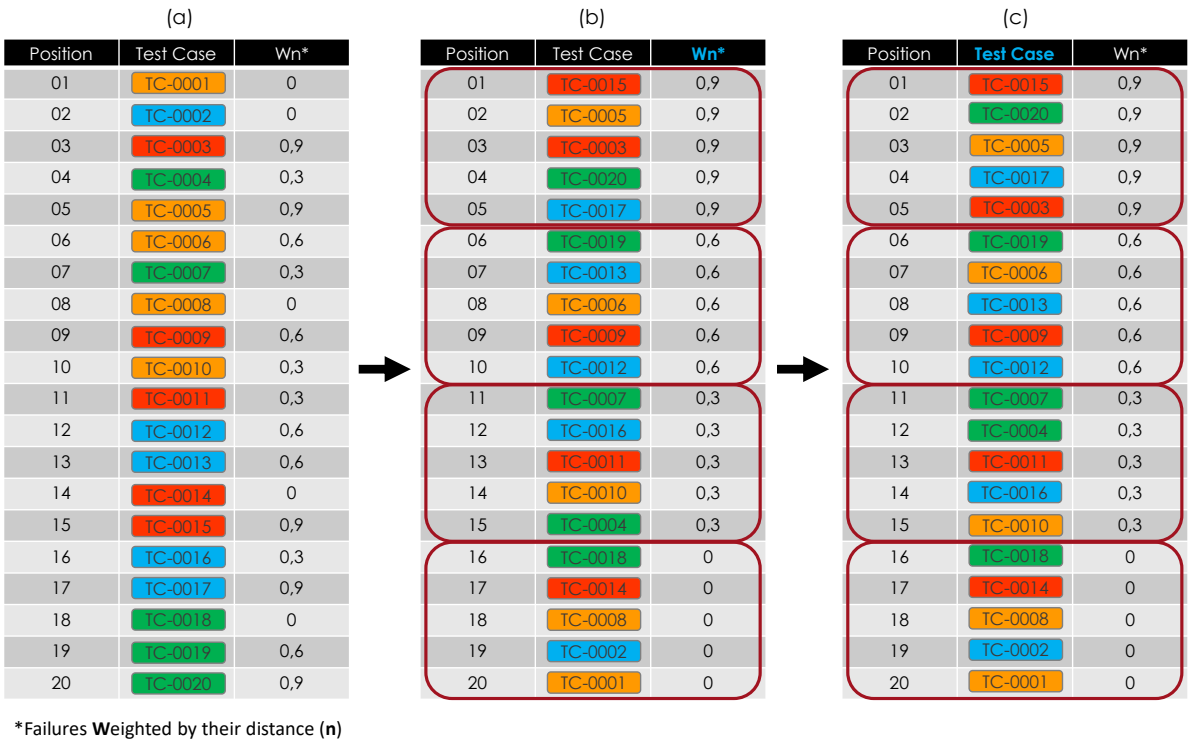
Position	Test Case	Wn*
01	TC-0015	0,9
02	TC-0020	0,9
03	TC-0005	0,9
04	TC-0017	0,9
05	TC-0003	0,9
06	TC-0019	0,6
07	TC-0009	0,6
08	TC-0013	0,6
09	TC-0006	0,6
10	TC-0012	0,6
11	TC-0007	0,3
12	TC-0011	0,3
13	TC-0016	0,3
14	TC-0010	0,3
15	TC-0004	0,3
16	TC-0018	0
17	TC-0014	0
18	TC-0008	0
19	TC-0002	0
20	TC-0001	0

*Failures **W**eighted by their distance (**n**)

Source: Created by the author

- History-Based Random (*HBR*): HBR is analogous to the previously described HBD. The only difference is that the intra-cluster tests (Figure 11 (b)) are not ordered based on distance metrics. Instead, the intra-clusters are simply shuffled (Figure 11 (c)).

Figure 11 – History-Based Random Ordering



Source: Created by the author

Notice that the so-called *Optimal prioritization* has no practical use, given that it is impossible to know which test cases would reveal which faults in advance. For the purpose of comparison, however, it is very useful, given that it allows us to assess how far a given technique is from the best possible result.

5.3.4 Experimental Object

The experimental objects are the two history-based approaches, family-dependent and family-independent, proposed in this study:

- Family-dependent (*Fam-dep*): the test suite, Figure 12 (a), is ordered based on the execution history that considers only the reported results for products that belong to the same family as the target product. First, we order the test cases in descending order considering their *failure frequency* with respect to the history of products that belong to the same family, represented in Figure 12 (b) column *FF-Fd*. Then, cases of ties are resolved considering the *test case age*, putting first the **new** and **updated** test cases, Figure 12 (c).
- Family-independent (*Fam-ind*): the test suite, Figure 13 (a), is ordered based on the execution history that considers the entire execution history of a test case, regardless of the product or software version for which the test case was executed. This approach is very similar to the previous one. We order the test cases in descending order, considering their *failure frequency*. For this approach, however, we consider the entire history of executions, regardless of the product's family (Figure 13 (b) column *FF-Fi*). Cases of ties are resolved as in the previous approach considering the *test case age*, putting first the **new** and **updated** test cases, Figure 13 (c).

Figure 12 – Family-dependent Ordering

(a)

Position	Test Case	Test Age	FF-Fi[1]	FF-Fd[2]
01	TC-0001	01/01/16	0,000	0,000
02	TC-0002	01/04/16	0,000	0,000
03	TC-0003	01/07/16	0,568	0,489
04	TC-0004	01/10/16	0,100	0,087
05	TC-0005	01/01/17	0,598	0,325
06	TC-0006	01/04/17	0,333	0,132
07	TC-0007	01/07/17	0,123	0,078
08	TC-0008	01/10/17	0,000	0,000
09	TC-0009	01/01/18	0,333	0,426
10	TC-0010	01/04/18	0,100	0,084
11	TC-0011	01/07/18	0,100	0,093
12	TC-0012	01/10/18	0,325	0,282
13	TC-0013	01/01/19	0,356	0,423
14	TC-0014	01/04/19	0,001	0,000
15	TC-0015	01/07/19	0,598	0,265
16	TC-0016	01/10/19	0,115	0,105
17	TC-0017	01/01/20	0,532	0,489
18	TC-0018	01/04/20	0,001	0,001
19	TC-0019	01/07/20	0,365	0,154
20	TC-0020	01/10/20	0,564	0,365

(b)

Position	Test Case	Test Age	FF-Fi[1]	FF-Fd[2]
01	TC-0003	01/07/16	0,568	**0489**
02	TC-0017	01/01/20	0,532	**0489**
03	TC-0009	01/01/18	0,333	0,426
04	TC-0013	01/01/19	0,356	0,423
05	TC-0020	01/10/20	0,564	0,365
06	TC-0005	01/01/17	0,598	0,325
07	TC-0012	01/10/18	0,325	0,282
08	TC-0015	01/07/19	0,598	0,265
09	TC-0019	01/07/20	0,365	0,154
10	TC-0006	01/04/17	0,333	0,132
11	TC-0016	01/10/19	0,115	0,105
12	TC-0011	01/07/18	0,100	0,093
13	TC-0004	01/10/16	0,100	0,087
14	TC-0010	01/04/18	0,100	0,084
15	TC-0007	01/07/17	0,123	0,078
16	TC-0018	01/04/20	0,001	0,001
17	TC-0001	01/01/16	0,000	**0,000**
18	TC-0002	01/04/16*	0,000	**0,000**
19	TC-0008	01/10/17	0,000	**0,000**
20	TC-0014	01/04/19	0,001	**0,000**

(c)

Position	Test Case	Test Age	FF-Fi[1]	FF-Fd[2]
01	TC-0017	**01/01/20**	0,532	**0489**
02	TC-0003	**01/07/16**	0,568	**0489**
03	TC-0009	01/01/18	0,333	0,426
04	TC-0013	01/01/19	0,356	0,423
05	TC-0020	01/10/20	0,564	0,365
06	TC-0005	01/01/17	0,598	0,325
07	TC-0012	01/10/18	0,325	0,282
08	TC-0015	01/07/19	0,598	0,265
09	TC-0019	01/07/20	0,365	0,154
10	TC-0006	01/04/17	0,333	0,132
11	TC-0016	01/10/19	0,115	0,105
12	TC-0011	01/07/18	0,100	0,093
13	TC-0004	01/10/16	0,100	0,087
14	TC-0010	01/04/18	0,100	0,084
15	TC-0007	01/07/17	0,123	0,078
16	TC-0018	01/04/20	0,001	0,001
17	TC-0014	**01/04/19**	0,001	**0,000**
18	TC-0002	**01/04/16***	0,000	**0,000**
19	TC-0008	**01/10/17**	0,000	**0,000**
20	TC-0001	**01/01/16**	0,001	**0,000**

[1]FF – Family Independent | [2]FF – Family Dependent | * Updated

Source: Created by the author

Figure 13 – Family-independent Ordering

(a)

Position	Test Case	Test Age	FF-Fi[1]	FF-Fd[2]
01	TC-0001	01/01/16	0,000	0,000
02	TC-0002	01/04/16	0,000	0,000
03	TC-0003	01/07/16	0,568	0,489
04	TC-0004	01/10/16	0,100	0,087
05	TC-0005	01/01/17	0,598	0,325
06	TC-0006	01/04/17	0,333	0,132
07	TC-0007	01/07/17	0,123	0,078
08	TC-0008	01/10/17	0,000	0,000
09	TC-0009	01/01/18	0,333	0,426
10	TC-0010	01/04/18	0,100	0,084
11	TC-0011	01/07/18	0,100	0,093
12	TC-0012	01/10/18	0,325	0,282
13	TC-0013	01/01/19	0,356	0,423
14	TC-0014	01/04/19	0,001	0,000
15	TC-0015	01/07/19	0,598	0,265
16	TC-0016	01/10/19	0,115	0,105
17	TC-0017	01/01/20	0,532	0,489
18	TC-0018	01/04/20	0,001	0,001
19	TC-0019	01/07/20	0,365	0,154
20	TC-0020	01/10/20	0,564	0,365

(b)

Position	Test Case	Test Age	FF-Fi[1]	FF-Fd[2]
01	TC-0005	01/01/17	**0,598**	0,325
02	TC-0015	01/07/19	**0,598**	0,265
03	TC-0003	01/07/16	0,568	0,489
04	TC-0020	01/10/20	0,564	0,365
05	TC-0017	01/01/20	0,532	0,489
06	TC-0019	01/07/20	0,365	0,154
07	TC-0013	01/01/19	0,365	0,423
08	TC-0006	01/04/17	**0,333**	0,132
09	TC-0009	01/01/18	**0,333**	0,426
10	TC-0012	01/10/18	0,325	0,282
11	TC-0007	01/07/17	0,123	0,078
12	TC-0016	01/10/19	0,115	0,105
13	TC-0004	01/10/16	**0,100**	0,087
14	TC-0010	01/04/18	**0,100**	0,084
15	TC-0011	01/07/18	**0,100**	0,093
16	TC-0014	01/04/19	**0,001**	0,000
17	TC-0018	01/04/20	**0,001**	0,001
18	TC-0001	01/01/16	**0,000**	0,000
19	TC-0002	01/04/16*	**0,000**	0,000
20	TC-0008	01/10/17	**0,000**	0,000

(c)

Position	Test Case	Test Age	FF-Fi[1]	FF-Fd[2]
01	TC-0015	**01/07/19**	**0,598**	0,265
02	TC-0005	**01/01/17**	**0,598**	0,325
03	TC-0003	01/07/16	0,568	0,489
04	TC-0020	01/10/20	0,564	0,365
05	TC-0017	01/01/20	0,532	0,489
06	TC-0019	01/07/20	0,365	0,154
07	TC-0013	01/01/19	0,365	0,423
08	TC-0009	**01/01/18**	**0,333**	0,426
09	TC-0006	**01/04/17**	**0,333**	0,132
10	TC-0012	01/10/18	0,325	0,282
11	TC-0007	01/07/17	0,123	0,078
12	TC-0016	01/10/19	0,115	0,105
13	TC-0011	**01/07/18**	**0,100**	0,093
14	TC-0010	**01/04/18**	**0,100**	0,084
15	TC-0004	**01/10/16**	**0,100**	0,087
16	TC-0018	**01/04/20**	**0,001**	0,001
17	TC-0014	**01/04/19**	**0,001**	0,000
18	TC-0008	**01/10/17**	**0,000**	0,000
19	TC-0002	**01/04/16***	**0,000**	0,000
20	TC-0001	**01/01/16**	**0,000**	0,000

[1]FF – Family Independent | [2]FF – Family Dependent | * Updated

Source: Created by the author

5.3.5 Independent Variables

The independent variables of our study are the implementation of the prioritization approaches as well as the tools used to collect and analyze data.

5.3.6 Dependent Variables

The APFD obtained after running the test suites and the total prioritization time.

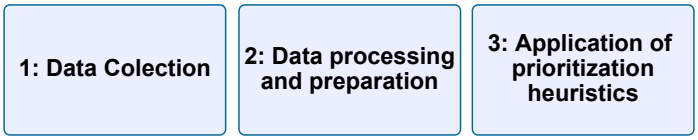
5.3.7 Trials Design

To answer research questions RQ1 and RQ2 our proposed history-based prioritization techniques as well as the alternative prioritization strategies were applied to the historical data we collected and their results were stored to be analyzed with respect to: (1) the APFD value obtained after each prioritization and (2) the total time required for completing the prioritization task. After collecting these results, statistical tests were applied to validate our hypotheses. Results are presented and discussed in Section 5.8.

5.4 PREPARATION

Figure 14 shows an overview of the preparation phase of our experimental study divided into three main tasks: (1) data collection, (2) data processing and preparation, and (3) the application of the prioritization heuristics.

Figure 14 – Heuristics Prioritization Process



Source: Created by the author

As part of the data collection task (1), to carry out our study, we collected historical execution data for 79 test suites created to validate 35 products from two families. The test suites contain a total of 21,072 test cases, and we processed a total of 5,859,989 test execution results. For the data processing and preparation (2), initially, we pre-processed the test case

execution data retrieved from the BigQuery. During this preparation, part of the data was filtered, keeping only the essential data needed for applying the APFD function. For example, for the issues collected, we verify whether it is a valid issue in the bug tracking tool. For the task number three in Figure 14, we use all the prioritization strategies that are part of the experimental and control objects for creating prioritized test suites that will be later used for computing the metrics adopted in our study.

5.5 ANALYSIS

The analysis is carried out slightly differently for the two RQs. For RQ1, we assess the effectiveness of the proposed history-based prioritization heuristics (experimental objects) when compared with alternative prioritization strategies (control objects) in terms of APFD. For RQ2, we compare the efficiency of the proposed prioritization heuristics against *state-of-the-art* competitors' heuristics in terms of the total time required for the prioritization.

As we have a one-factor study with data that does not follow a normal distribution, confirmed with the Anderson-Darling normality test (ANDERSON; DARLING, 1952), we use a Kruskal-Wallis rank sum test to analyze the results. The Kruskal-Wallis test is a non-parametric statistical method for testing whether samples originate from the same distribution and is used to compare two or more independent samples of equal or different sizes (KRUSKAL; WALLIS, 1952). We complement the Kruskal-Wallis test result with the Vargha and Delaney's A (VD.A) metric of effect size (VARGHA; DELANEY, 2000) to assess the magnitude of the observed differences.

5.6 THREATS TO VALIDITY

This section discusses how valid the results are and whether we can generalize them to a larger population.

- **Internal Validity.** One threat we can observe is in selecting the suites that will be used for the study, as different test suites of different products could have been selected. Control for this threat can only be achieved by running additional experiments. Another threat that is worth mentioning is that the number of faults that can be identified by a given test suite can influence the maximum value that can be achieved by the APFD

metric. Additional internal validity threats of this work include the selection bias, the history used, and the sample size.

- **External Validity.** We cannot claim the generalization of our results beyond the particular context where our study was conducted. As our experiments were carried out within a specific industrial environment, our findings are limited by the products we used in our study. To minimize this threat, we need to conduct additional experiments using different products and, if possible, in different industrial settings.

- **Construct Validity.** As the results were collected directly from the test suites that were generated, greater concerns about how to collect the data were not necessary. Nevertheless, we should take into account that these suites have different sizes and were used for different products.

- **Conclusion Validity.** One common conclusion validity threat is the violation of the assumptions of the statistical methods used. For our study, we consider such a threat to be low as we used a non-parametric test, the Kruskal-Wallis rank sum test, which does not rely on assumptions about the distribution of the data. Although the observed significant results in our study are in line with our expectations, additional studies using different subjects should be conducted to minimize this threat.

5.7 EXECUTION

To answer RQ1 and RQ2, we applied the proposed approach and the alternative prioritization strategies to each experimental subject and measured APFD and prioritization time. To account for the stochastic nature of the random prioritization, we considered the average APFD value obtained after 30 repetitions.

5.8 RESULTS

In this section we report and discuss the results. With the aim of supporting the independent verification of our results, we make available the artefacts we produced for conducting this study[2]. Our supplementary material (SIQUEIRA; MIRANDA, 2022b) includes the implementation

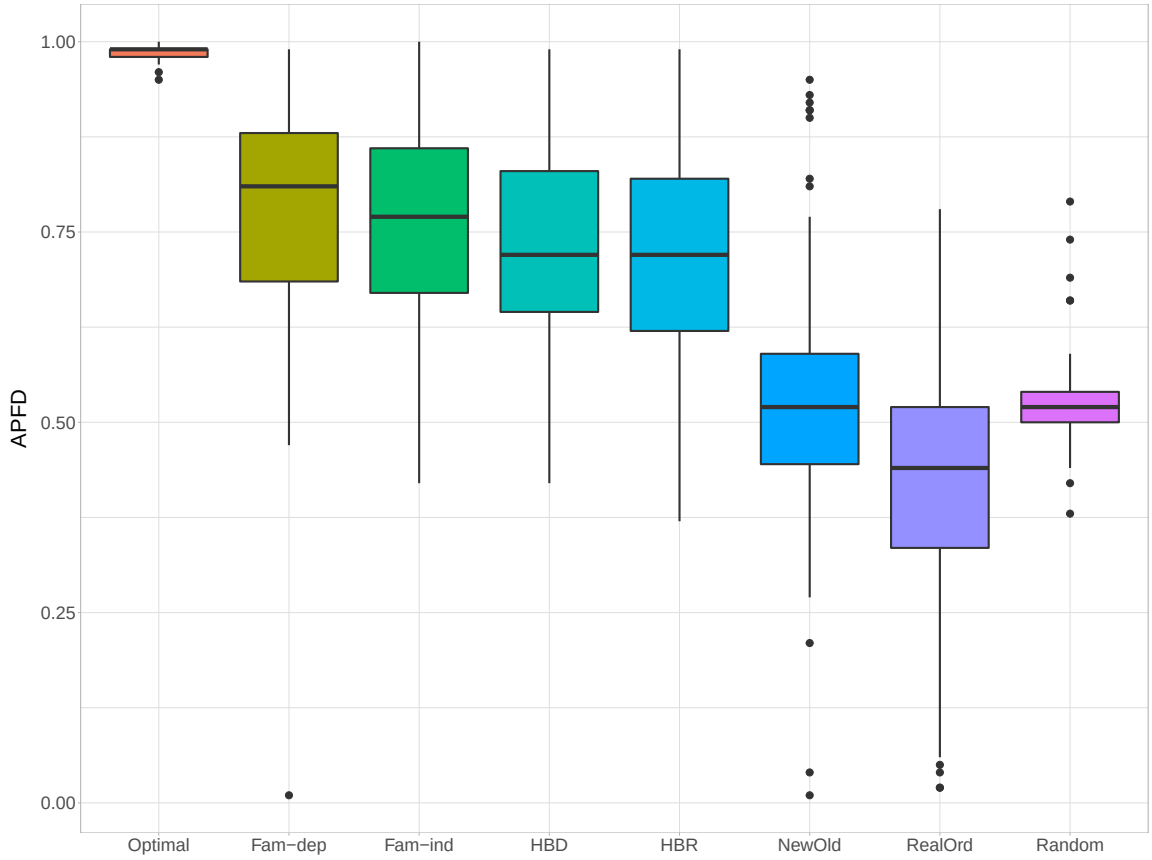
[2] https://github.com/HBPrio/msc2022

of the proposed heuristics, the scripts used for running the experiments, and the scripts used for running the statistical analysis.

5.8.1 RQ1: Effectiveness

The APFD results achieved for each prioritization strategy are displayed as box plots in Figure 15. The visual assessment of the data shows us that the *Optimal Prioritization*, as expected, achieves the maximum median (0.99), followed by our two proposed heuristics *Family-dependent* (0.81), and *Family-independent* (0.77). *State-of-the-art* competitors come next, with competitors *HBD* (0.72) and *HBR* (0.72) tied. The ordering suggested by the *Default Prioritization* (0.52) and the *Real Prioritization* order followed by the testers (0.44) achieved a median APFD equal to (or worse than) that of a *Random Prioritization* (0.52).

Figure 15 – Box-plot of APFD Results for all Heuristics



Source: Created by the author

After the visual assessment, we proceeded with the statistical analysis of the data. As we could not assume our data to be normally distributed[3], we use the non-parametric Kruskal-

[3] Confirmed with the Anderson-Darling normality test (ANDERSON; DARLING, 1952).

Wallis rank sum test to assess, at a significance level of 5%, the null hypothesis that the differences in the APFD values for the different TCP approaches are not statistically significant. The observed differences in APFD were statistically significant ($p - value < 2.2e - 16$).

A significant Kruskal-Wallis rank sum test only indicates that at least one of the TCP heuristics has statistical dominance over the others, without identifying which pairs of approaches are different. Because of that, we performed pairwise comparisons to determine which TCP approaches are different. The results are shown in the column *Group* in Table 2.

Table 2 – RQ1: Pairwise Comparisons

Approach	*Med*	*SD*	*Group*
Optimal	0.99	0.01	(a)
Fam-dep	0.81	0.16	(b)
Fam-ind	0.77	0.13	(bc)
HBD	0.72	0.14	(cd)
HBR	0.72	0.15	(d)
NewOld	0.52	0.18	(e)
Random	0.52	0.06	(e)
RealOrd	0.44	0.16	(f)

Med is the APFD median, *SD* is the standard deviation, and *Group* displays the result for the pairwise comparisons after the Kruskal-Wallis test.

Approaches with different letters are significantly different, whereas the difference between the approaches with the same letter is not statistically significant. The approach that yields the best performance is assigned to the **group (a)**.

Analyzing the results in Table 2, we can see that *Optimal (a)*, as expected, is different from (better than) all the other strategies. An approach can have more than one letter assigned to it. As an example, looking at the results in Table 2, we can tell that *Fam-ind (bc)* is not different from *Fam-dep (b)* and it is also not different from *HBD (cd)*, even though *Fam-dep (b)* is different from (better than) *HBD (cd)*. *HBR (d)* is outperformed by *Optimal (a)*, *Fam-dep (b)*, and *Fam-ind (bc)*, but it can be as good as *HBD (cd)*. No statistical difference can be observed between the ordering suggested by the *Default Prioritization* (*NewOld*) and the *Random Prioritization (e)*, and they are both different (better than) from the real execution order followed by the testers *RealOrd (f)*.

Looking from the perspective of the proposed approach, when we consider the median APFD, the *Family-dependent* heuristic performed slightly better than the *Family-independent*

one. This is in agreement with our intuition, given that the family-dependent approach considers only the history of test cases that were executed on similar products.

A statistical test will yield a value of p (which is a value of probability), and based on this $p-value$, we can decide whether the observed differences are statistically significant or not; however, the $p-value$ does not reflect the magnitude of the observed effect. After analyzing the Kruskal-Wallis test results, we also apply the *Vargha and Delaney's A* (VD.A) measure (VARGHA; DELANEY, 2000) of effect size to assess the magnitude of the observed differences. The VD.A results can be observed in Table 3 and they are provided as pairwise comparisons between all the evaluated approaches. Next we discuss the most relevant observations for the context of our study:

- The (*Optimal*) prioritization has a **Large** effect size over all the other approaches (expected).
- One of the proposed heuristics (*Fam-dep*) is significantly better than the two *state-of-the-art* approaches investigated (*HBD* and *HBR*), although it was by a **small** effect size.
- When we compare our proposed approaches with each other (*Fam-dep* x *Fam-ind*), we can observe a **Negligible** difference between them. This is in line with the result of the pairwise comparisons, as both approaches shared the same group: *Fam-dep (b)* and *Fam-ind (bc)*.
- When we compare our proposed approaches (*Fam-dep & Fam-ind*) against the *Random* prioritization, the test management tool (*NewOld*), and the real execution order followed by the testers (*RealOrd*), we also observe a **Large** effect size.

Table 3 – RQ1: VD.A Effect Size

Comparison	*CD*	*rg*	*VDA.m*	*Effect Size**
Optimal x Fam-dep	0.968	0.967	0.984	Large
Optimal x Fam-ind	0.970	0.971	0.985	Large
Optimal x HBD	0.962	0.962	0.981	Large
Optimal x HBR	0.966	0.966	0.983	Large
Optimal x Random	1.000	1.000	1.000	Large
Optimal x NewOld	1.000	1.000	1.000	Large
Optimal x RealOrd	1.000	1.000	1.000	Large
Fam-dep x Fam-ind	0.110	0.110	0.555	Negligible
Fam-dep x HBD	0.188	0.188	0.594	Small
Fam-dep x HBR	0.236	0.237	0.618	Small
Fam-dep x Random	0.846	0.847	0.923	Large
Fam-dep x NewOld	0.716	0.716	0.858	Large
Fam-dep x RealOrd	0.896	0.897	0.948	Large
Fam-ind x HBD	0.100	0.101	0.550	Negligible
Fam-ind x HBR	0.158	0.158	0.579	Small
Fam-ind x Random	0.850	0.851	0.925	Large
Fam-ind x NewOld	0.708	0.709	0.854	Large
Fam-ind x RealOrd	0.906	0.906	0.953	Large
HBD x HBR	0.064	0.064	0.532	Negligible
HBD x Random	0.802	0.802	0.901	Large
HBD x NewOld	0.660	0.659	0.830	Large
HBD x RealOrd	0.876	0.876	0.938	Large
HBR x Random	0.744	0.745	0.872	Large
HBR x NewOld	0.608	0.609	0.804	Large
HBR x RealOrd	0.834	0.835	0.917	Large
Random x NewOld	0.008	0.008	0.504	Negligible
Random x RealOrd	0.442	0.443	0.721	Large
NewOld x RealOrd	0.352	0.352	0.676	Medium

CD: ranges from -1 to 1, with 0 indicating stochastic equality, and 1 indicating that the first group dominates the second.
rg: ranges from -1 to 1, depending on sample size, with 0 indicating no effect, and a positive result indicating that values in the first group are greater than in the second.
VD.A: ranges from 0 to 1, with 0.5 indicating stochastic equality, and 1 indicating that the first group dominates the second.

**VD.A: Vargha and Delaney A measure (Negligible, <=0.55 | Small, >= 0.56 | Medium, >= 0.64 | Large, >= 0.71).*

Answer to RQ1

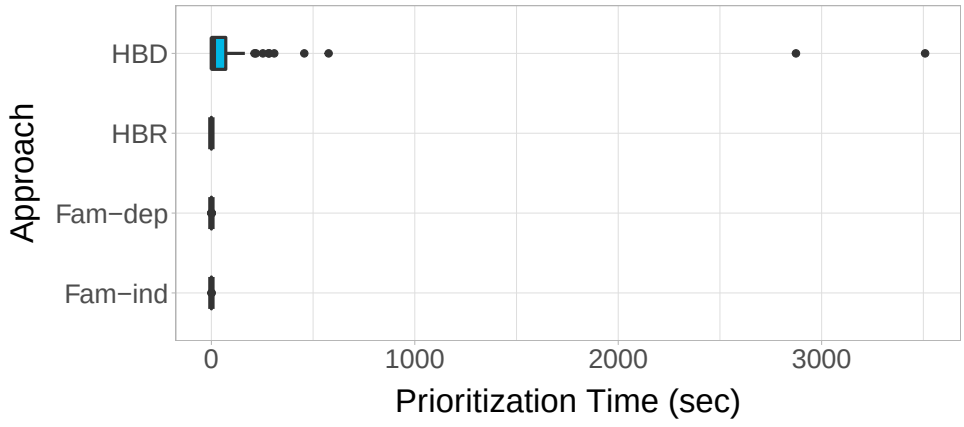
The proposed prioritization heuristics were shown to be highly effective, achieving APFD values that are not so far from those achieved by an optimal ordering. One of the proposed heuristics (Fam-dep) is statistically significantly better than the two state-of-the-art approaches investigated (HBD and HBR), although it was by a small effect size. The two proposed heuristics are significantly better than the two approaches currently in use in the company's context (the test management tool suggestion and the tester's own choice), with a large effect size.

5.8.2 RQ2: Efficiency

To answer RQ2, we analyzed the total prioritization time required by the proposed approaches. Similarly to what was done for answering RQ1, we start with the visual inspection of the box plots displayed in Figure 16. For this metric, however, the lower the value, the better.

Given the high prioritization times required by *HBD* (median is 132.90), in Figure 16 we cannot really appreciate the results produced by the other approaches, thus we produced another set of boxplots without including the *HBD* approach (Figure 17).

Figure 16 – Box-plot of Prioritization Time Results for all Heuristics



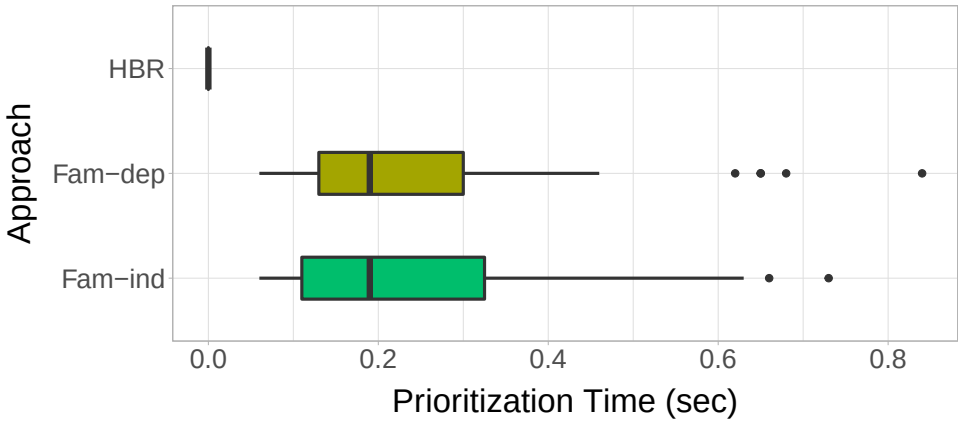
Source: Created by the author

In Figure 17 we can observe that *HBR* is very efficient, with a median prioritization time of 0.0023 milliseconds. This happens because *HBR* produces its prioritized test suite simply by shuffling the list of test cases.

When we consider the proposed heuristics, we can see that both approaches had very

similar performance, with the median time of *family-independent* (0.17) being only slightly better than that of the *family-dependent* heuristic (0.20) (Figure 17).

Figure 17 – Box-plot of Prioritization Time Results for Proposed Heuristics and HBR



Source: Created by the author

We then proceeded with the statistical analysis of the data. Given that our data was not normally distributed, we performed again the Kruskal-Wallis rank sum test. For RQ2 we assess, at a significance level of 5%, the null hypothesis that the differences in the total prioritization time for the proposed heuristics are not statistically significant.

The observed differences in total prioritization time were statistically significant with $(p{-}value < 2.2e-16)$. As stated before, a significant Kruskal-Wallis rank sum test only indicates that at least one of the TCP heuristics has statistical dominance over the others, without identifying which pairs of approaches are different. Because of that, we performed pairwise comparisons to determine which TCP approaches are different. The results are shown in the column *Group* in Table 4.

Table 4 – RQ2: Pairwise Comparisons

Approach	*Med*	*SD*	*Group*
HBD	132.90	509.64	(a)
Fam-dep	0.20	0.16	(b)
Fam-ind	0.17	0.16	(b)
HBR	0.00	0.00	(c)

Med is the Prioritization time median, *SD* is the standard deviation, and *Group* displays the result for the pairwise comparisons after the Kruskal-Wallis test.

For interpreting Table 4, recall that approaches with different letters are significantly different, whereas the difference between the approaches with the same letter is not statistically

significant. For this RQ, the approach that yields the worst performance is assigned to the **group (a)**.

Analyzing the results in Table 4, we can see that *HBD (a)* is different from (worst than) *Family-dependent (b)*, which in turn is tied with *Family-independent (b)*, meaning that no statistical difference can be observed between the proposed approaches with respect to prioritization time. *HBR* is isolated in group *(c)* with the best prioritization time.

Table 5 – RQ2: VD.A Effect Size

Comparison	*CD*	*rg*	*VDA.m*	*Effect Size*
Fam-ind x Fam-dep	0.006	0.007	0.503	Negligible
Fam-ind x HBD	-0.830	-0.830	0.915	Large
Fam-ind x HBR	1.000	1.000	1.000	Large
Fam-dep x HBD	-0.823	-0.823	0.912	Large
Fam-dep x HBR	1.000	1.000	1.000	Large
HBD x HBR	1.000	1.000	1.000	Large

CD: ranges from -1 to 1, with 0 indicating stochastic equality, and 1 indicating that the first group dominates the second.
rg: ranges from -1 to 1, depending on sample size, with 0 indicating no effect, and a positive result indicating that values in the first group are greater than in the second.
VD.A: ranges from 0 to 1, with 0.5 indicating stochastic equality, and 1 indicating that the first group dominates the second.

**VD.A: Vargha and Delaney A measure (Negligible, <=0.55 | Small, >= 0.56 | Medium, >= 0.64 | Large, >= 0.71).*

We also applied the *Vargha and Delaney's A* (VD.A) measure of effect size to complement the Kruskal-Wallis test result. The VD.A results can be observed in Table 5. The main observation is that the effect size of the pairwise comparison for the proposed approaches is *negligible*.

Answer to RQ2

The proposed prioritization heuristics are highly efficient, being able to complete the prioritization task in less than 1 second (for an average test suite with ≈ 1500 test cases). HBR is very efficient given that its prioritized test suite is produced simply by shuffling the list of test cases. Such a strategy, however, trades effectiveness for efficiency. The version HBD was outperformed by our proposed strategies with a large effect size. Despite the different number of test results processed by the proposed heuristics, no statistically significant differences were observed. This happens because, thanks to the infrastructure used for storing the test results, the query time is virtually the same regardless of the number of results returned.

5.9 CONCLUDING REMARKS

In this chapter, we described the experimental study that evaluated the effectiveness and efficiency of the proposed heuristics, and we discussed the results of our experiment. In terms of effectiveness, the proposed prioritization heuristics were shown to be highly effective, achieving APFD values that are not so far from those achieved by an optimal ordering. One of the proposed heuristics (*Fam-dep*) is significantly better than the two *state-of-the-art* approaches investigated. On top of that, the two proposed heuristics are significantly better than the two currently used approaches in the company's context. With respect to efficiency, the proposed prioritization heuristics are highly efficient, being able to complete the prioritization task in less than 1 second (for an average test suite with ≈ 1500 test cases).

6 CONCLUSION AND FUTURE WORK

In this work, we introduced two history-based prioritization heuristics and evaluated them in the context of manual testing in a real industrial setting. We compared our proposed approaches against alternative prioritization strategies, including a *state-of-the-art* history-based approach, an optimal prioritization, the real ordering followed by the testers, the ordering suggested by the test management tool, and a random ordering. For our evaluation, we collected historical test execution information from 35 products, spanning over seven years of historical information, accounting for a total of 3,196 unique test cases and 5,859,989 test results.

MAIN FINDINGS

The results of our experiments using historical test execution data from real subjects and with real faults support the adoption of our approach. They showed that the effectiveness of the proposed heuristics is not far from a theoretical optimal prioritization and that they are significantly better than alternative orderings of the test suite, including *state-of-the-art* approaches, the order suggested by the test management tool, and the execution order followed by the testers during the real execution of the test suites evaluated as part of our study. With respect to efficiency, our proposed approaches yield similar results, and they are both better (faster) than one of the *state-of-the-art* history-based competitors.

IMPLICATIONS

The results of our experiments provide actionable information to the test managers of the company where we conducted our studies: the order followed by the testers in real life achieved the worst results, and even the order suggested by the test management tool is only as good as a random prioritization. Adopting our proposed approach could be the first step towards the goal of improving the company's testing effectiveness.

FUTURE WORK

As part of our future work, we plan to conduct additional experiments to assess the effectiveness and efficiency of our proposed approach when compared with existing history-based prioritization strategies — in addition to the *state-of-the-art* approach investigated in this work. It was out of the scope of this work to consider dependencies between test cases (e.g., test cases that should be run before/after another test case because of setup reuse), but we plan to investigate such dependencies as part of our future work.

REFERENCES

ANDERSON, T. W.; DARLING, D. A. Asymptotic theory of certain" goodness of fit" criteria based on stochastic processes. *The annals of mathematical statistics*, JSTOR, p. 193–212, 1952.

BAJAJ, A.; SANGWAN, O. P. A systematic literature review of test case prioritization using genetic algorithms. *IEEE Access*, IEEE, v. 7, p. 126355–126375, 2019.

BASILI, V. R. Goal question metric paradigm. *Encyclopedia of software engineering*, John Weily and Sons, p. 528–532, 1994.

BERTOLINO, A.; GUERRIERO, A.; MIRANDA, B.; PIETRANTUONO, R.; RUSSO, S. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. [S.l.: s.n.], 2020. p. 1–12.

BERTOLINO, A.; MIRANDA, B.; PIETRANTUONO, R.; RUSSO, S. Adaptive test case allocation, selection and generation using coverage spectrum and operational profile. *IEEE Transactions on Software Engineering*, IEEE, v. 47, n. 5, p. 881–898, 2019.

CATAL, C.; MISHRA, D. Test case prioritization: a systematic mapping study. *Software Quality Journal*, Springer, v. 21, n. 3, p. 445–478, 2013.

CRUCIANI, E.; MIRANDA, B.; VERDECCHIA, R.; BERTOLINO, A. Scalable approaches for test suite reduction. In: IEEE. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. [S.l.], 2019. p. 419–429.

DIJKSTRA, E. W. et al. *Notes on structured programming*. [S.l.]: Technological University, Department of Mathematics, 1970.

DO, H. Recent advances in regression testing techniques. *Advances in computers*, Elsevier, v. 103, p. 53–77, 2016.

ELBAUM, S.; KALLAKURI, P.; MALISHEVSKY, A.; ROTHERMEL, G.; KANDURI, S. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software testing, verification and reliability*, Wiley Online Library, v. 13, n. 2, p. 65–83, 2003.

ELBAUM, S.; ROTHERMEL, G.; PENIX, J. Techniques for improving regression testing in continuous integration development environments. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014. p. 235–245.

ENGSTRÖM, E.; RUNESON, P.; LJUNG, A. Improving regression testing transparency and efficiency with history-based prioritization–an industrial case study. In: IEEE. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. [S.l.], 2011. p. 367–376.

ENGSTRÖM, E.; RUNESON, P.; SKOGLUND, M. A systematic review on regression test selection techniques. *Information and Software Technology*, Elsevier, v. 52, n. 1, p. 14–30, 2010.

FAZLALIZADEH, Y.; KHALILIAN, A.; AZGOMI, M. A.; PARSA, S. Incorporating historical test case performance data and resource constraints into test case prioritization. In: SPRINGER. *International conference on tests and proofs*. [S.l.], 2009. p. 43–57.

GRAVES, T. L.; HARROLD, M. J.; KIM, J.-M.; PORTER, A.; ROTHERMEL, G. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 10, n. 2, p. 184–208, 2001.

GRECA, R.; MIRANDA, B.; GLIGORIC, M.; BERTOLINO, A. Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration. In: *3rd ACM/IEEE International Conference on Automation of Software Test*. [S.l.: s.n.], 2022.

HAAS, R.; ELSNER, D.; JUERGENS, E.; PRETSCHNER, A.; APEL, S. How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens, Greece: ACM, 2021. p. 1281–1291.

HAGHIGHATKHAH, A.; MÄNTYLÄ, M.; OIVO, M.; KUVAJA, P. Test prioritization in continuous integration environments. *Journal of Systems and Software*, Elsevier, v. 146, p. 80–98, 2018.

HAO, D.; ZHANG, L.; MEI, H. Test-case prioritization: achievements and challenges. *Frontiers of Computer Science*, Springer, v. 10, n. 5, p. 769–777, 2016.

HUANG, Y.-C.; PENG, K.-L.; HUANG, C.-Y. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, Elsevier, v. 85, n. 3, p. 626–637, 2012.

IEEE. Ieee standard for software test documentation. *IEEE Std 829-1983*, p. 1–48, 1983.

IEEE. Ieee standard for software and system test documentation. *IEEE Std 829-2008*, p. 1–150, 2008.

IEEE. Ieee standard for system, software, and hardware verification and validation - redline. *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017) - Redline*, p. 1–465, 2017.

JEFFREY, D.; GUPTA, N. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on software Engineering*, IEEE, v. 33, n. 2, p. 108–123, 2007.

JURISTO, N.; MORENO, A. M. *Basics of software engineering experimentation*. [S.l.]: Springer Science & Business Media, 2013.

KHAN, S. U. R.; LEE, S. P.; JAVAID, N.; ABDUL, W. A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines. *IEEE Access*, IEEE, v. 6, p. 11816–11841, 2018.

KHATIBSYARBINI, M.; ISA, M. A.; JAWAWI, D. N.; TUMENG, R. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, Elsevier, v. 93, p. 74–93, 2018.

KIM, J.-M.; PORTER, A. A history-based test prioritization technique for regression testing in resource constrained environments. In: *Proceedings of the 24th international conference on software engineering*. [S.l.: s.n.], 2002. p. 119–129.

KRUSKAL, W. H.; WALLIS, W. A. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, Taylor & Francis, v. 47, n. 260, p. 583–621, 1952.

LIMA, L. A. d. *Test case prioritization based on data reuse for black-box environments*. Master's Thesis (Master's Thesis) — Universidade Federal de Pernambuco, 2009.

MIRANDA, B.; BERTOLINO, A. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software*, Elsevier, v. 131, p. 528–549, 2017.

MIRANDA, B.; CRUCIANI, E.; VERDECCHIA, R.; BERTOLINO, A. Fast approaches to scalable similarity-based test case prioritization. In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. [S.l.], 2018. p. 222–232.

MUKHERJEE, R.; PATNAIK, K. S. A survey on different approaches for software test case prioritization. *Journal of King Saud University-Computer and Information Sciences*, Elsevier, 2018.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. [S.l.]: John Wiley & Sons, 2011.

NAJAFI, A.; SHANG, W.; RIGBY, P. C. Improving test effectiveness using test executions history: An industrial experience report. In: IEEE. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. [S.l.], 2019. p. 213–222.

NARDO, D. D.; ALSHAHWAN, N.; BRIAND, L.; LABICHE, Y. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 25, n. 4, p. 371–396, 2015.

NOOR, T. B.; HEMMATI, H. A similarity-based approach for test case prioritization using historical failure data. In: IEEE. *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.], 2015. p. 58–68.

ORSO, A.; SHI, N.; HARROLD, M. J. Scaling regression testing to large software systems. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 29, n. 6, p. 241–251, 2004.

RIOS, E.; MOREIRA, T. *Teste de software*. [S.l.]: Alta Books Editora, 2006.

ROTHERMEL, G.; HARROLD, M. J. A safe, efficient algorithm for regression test selection. In: IEEE. *1993 Conference on Software Maintenance*. [S.l.], 1993. p. 358–367.

ROTHERMEL, G.; HARROLD, M. J. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, IEEE, v. 22, n. 8, p. 529–551, 1996.

ROTHERMEL, G.; UNTCH, R. H.; CHU, C.; HARROLD, M. J. Test case prioritization: An empirical study. In: IEEE. *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. [S.l.], 1999. p. 179–188.

SIQUEIRA, V.; MIRANDA, B. Investigating the adoption of history-based prioritization in the context of manual testing in a real industrial setting. In: IEEE. *2022 48th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA)*. [S.l.], 2022.

SIQUEIRA, V.; MIRANDA, B. *Supplementary material: Investigating the Adoption of History-based Prioritization in the Context of Manual Testing in a Real Industrial Setting*. 2022. <https://github.com/HBPrio/mscdis2022>. Accessed: 2022-04-29.

SRIKANTH, H.; COHEN, M. B.; QU, X. Reducing field failures in system configurable software: Cost-based prioritization. In: IEEE. *2009 20th International Symposium on Software Reliability Engineering*. [S.l.], 2009. p. 61–70.

VARGHA, A.; DELANEY, H. D. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, Sage Publications Sage CA: Los Angeles, CA, v. 25, n. 2, p. 101–132, 2000.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012.

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, Wiley Online Library, v. 22, n. 2, p. 67–120, 2012.