FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mining Web Usage to Generate Regression GUI Tests Automatically

Marta Inês Macedo Vasconcelos

Mestrado em Engenharia de Software

Supervisor: Ana Cristina Ramada Paiva

Second Supervisor: André Monteiro de Oliveira Restivo

July 19, 2019

Mining Web Usage to Generate Regression GUI Tests Automatically

Marta Inês Macedo Vasconcelos

Mestrado em Engenharia de Software

July 19, 2019

Abstract

Software systems tend to exceed expectations regarding their lifespan. During their time, evolution and maintenance might be needed, and regression testing must be performed to ensure behaviour is preserved. Gathering information from execution traces, when a software is in operation, may be useful in several contexts. In particular, execution traces may be used as a source of data for regression test case generation. However, selecting a subset of such traces that is representative and useful to detect problems when implementing changes may be a challenge. This paper presents an approach where user interactions on a website are automatically captured and analyzed to generate a test suite that could later be applied in regression testing. It starts by capturing usage data. The usage data obtained does not include information about the user or any input data to respect current data protection legislation. Then, the collected sessions are analyzed according to defined criteria in order to choose the subset of test cases that match the testing purpose of the application. At the end of the process, automated test scripts are generated. Finally, this paper presents a case study on a real website and some metrics that evaluate the quality of the generated test suite. The coverage of the application is also analyzed according to different metrics.

Keywords: regression testing, GUI testing, test case generation, software test automation, software testing

Resumo

A manutenção e a evolução de um sistema de *software* é essencial durante o seu tempo de vida. Para que as suas funcionalidades continuem a ser asseguradas, mesmo com alterações no producto, os testes de regressão desempenham um papel essencial. Recolher informação de utilização real do *software* pode ser útil em contextos diversos, em particular, no contexto de testes de regressão. A informação sobre a utilização pode ser usada como base para gerar testes de regressão a partir desses dados. No entanto, pode ser desafiante saber quais os subconjuntos de dados que são importantes para testar e quais os que são representativos para detectar falhas no sistema. Este trabalho propõe uma abordagem que recolhe dados reais a partir das interações feitas pelos utilizadores do *website* e as analisa de forma a gerar casos de testes para serem usados como testes de regressão. O processo inicia com a recolha de dados. Os dados recolhidos não contêm informação sensível sobre o utilizador nem dados introduzidos pelo mesmo, de forma a respeitar as atuais leis de proteção de dados. Depois de recolhidas as sessões e respectiva informação, estas são analisadas de acordo com alguns filtros e critérios, para tentar encontrar o conjunto de testes que mais se adequa ao *software* em causa. O processo culmina na geração de testes automáticos. Para concluir, é apresentado um caso de estudo, baseado num website real. A qualidade dos testes gerados é analisada segundo diversos factores, utilizando diferentes métricas para avaliar a cobertura da aplicação.

Keywords: Testes de regressão, Testes de interface gráfica, Geração de casos de teste, testes automáticos de *software*, testes de *software*

Acknowledgements

Firstly, I would like to thank my parents, for always supporting and believing in me. To my father, from whom I inherited the will to make so many things (from all the rest), thanks for showing me that 24h a day is enough to make more than most of the people do. To my mother, who always shows me a solution for everything, and who taught me that the most important thing is to give my best and to be happy. I'm sorry for all the weekends that I didn't come home. Big thanks, as big as my morning bad humour.

Thanks to all my friends for the continuous support in every moment of this journey: since the uncertain beginning to the busy ending. To my master's colleagues for the friendship during these two years. To Critical Software, and especially to my team, for the flexibility that made easier to conciliate everything.

Last but not least, I would like to thank my supervisor, Ana Paiva and second supervisor, André Restivo for the availability, commitment and pragmatism. Your continuous support was crucial to the success of this work. Thanks for every lunchtime you spent on meetings with me.

Marta Vasconcelos

"It's what you do in the dark that puts you in the light.
Rule yourself."

Under Armour Commercial (2016)

Contents

List of Figures

List of Tables

Abbreviations and Symbols

ISTQB	International Software Testing Qualifications Board
GUI	Graphical User Interface
UI	User Interface
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
UML	Unified Modeling Language
XML	Extensible Markup Language
DOM	Document Object Model
API	Application Programming Interface
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
HTTP	HyperText Transfer Protocol
VPN	Virtual Private Network
MARTT	Mining Automated Regression Testing Tool
$\bar{x}$	Arithmetic mean
σ	Standard deviation

Chapter 1

Introduction

This chapter introduces the context of this work in the first section (1.1). Then, the motivation and objectives are presented (1.2), followed by the definition of the problem and the research questions that this work aims to answer (1.3). Finally, the structure of the dissertation is described (1.4).

1.1 Context

Software has become increasingly complex and plays a vital role in our daily basis and in our society. Quality concerns have become essential in the software development process. In the end, the goal is to ensure that the system works as expected and provides an excellent experience to its users.

Web applications are usually composed by a back-end part and a front-end part, consumed by users through a browser. The back-end is responsible for the business logic and to process and persist data. Front-end, responsible for showing data and how users interact with the application, is becoming richer and more dynamic, allowing users to get custom data presentation according to the input made by them. This dynamic information leads to many challenges when testing Web applications.

Both parts, back-end and front-end, are usually implemented with different languages, dealing with many different technologies. Testing interaction among components built with different technologies can be a very challenging task. Nevertheless, this kind of applications became a demanding area of testing. They can be accessed from everywhere by a wide range of users: place, time zone, languages, disabilities, mindsets, cultures and so on. Users are also expecting high performance and reliability of the application. When it does not meet users' expectations, they look for another one. This leads to a highly competitive market that makes testing and quality control important factors of distinguishment [4].

It is estimated that software testing costs around 50% of the total development cost but, in some cases, it can reaches 80%. Test automation can be substantial to cost reduction, since it is usually faster, aiming to reduce the cost of testing in long-term, minimizing human error. Automated testing is more organized, structured and reproducible than manual testing, due to the detailed log of each test step and each activity, being easy to perceive where is the error, as well as to reproduce it [53, 47, 20, 39, 51, 4].

Although automated testing saves a lot of time when running test cases, it is important to have in mind that develop an automated test suite must be faced like a development project, which has to allocate enough time and resources to do so. The initial investment in automated testing is higher than on manual testing, either in building the testing scripts and training before using automated testing tools. However, the return on investment is also higher, saving a lot of time on a forward phase of the project, allowing to more frequent executions of a test suite, leading to early defects find. Teams must be available to highly invest in the initial phase of automated testing implementation, either with time or money. Because people often want immediate results, there is no availability to invest on that initial work, being automated tests pulled away from the development life-cycle [4, 20, 15].

Nowadays, to improve the user experience, most systems provide a Graphical User Interface (GUI) through which users interact with the system. System testing through the GUI is common and may be automated. However, despite all the benefits, automated GUI testing is not used as much as it could be. All of the reasons referred before as factors that lead to an automated testing avoidance also apply to automated GUI testing. However, maintenance has a special impact on GUI testing since it can be difficult to maintain test cases when minor changes, even only at the UI level, can lead to test breaking. [20].

Automated GUI testing has been studied, and many approaches have been developed to test systems through the GUI in the most efficient way. These approaches try to reduce the testing effort, making the test execution more consistent and test results more reliable. There are several approaches that try to adapt to each application's needs.

Automated testing is a good solution especially on regression testing context since previous features have to be re-tested, usually, with a major part of them without changes. Automating this process allows executing much more times the tests that would be possible to execute by doing it manually. Moreover, it is done with less effort and in a more efficient way, making regression testing easier [15].

1.2 Motivation and Objectives

Test automation has some challenges. Some of those are the lack of testing time, lack of enough allocated effort to build test scrips, and lack of sufficient test regression automation, which leads to weak quality control. Having regression testing is especially important when changes are made in the application or when a new version is deployed. Automated testing can save a lot of time and effort to make sure that new introduced features and updates did not break previously working functionalities, avoiding to test manually all the scenarios [20, 19, 9].

Actually, most of the studied automated approaches to generate test cases require some previous manual work: either model or documentation, test scenarios definition and simulation to be recorded or scripting development work. However, most of the times, applications are built not only without automated tests but also without any type of documentation, which prevents from any kind of test generation from models, specifications or other documentation. Since automated testing is not a priority for this kind of projects, there is no effort to manually develop test cases. This lack of documentation and testing do not allow to have proper quality control over the project, hinder the evolution process and the addition of new features.

Graphical User Interfaces have become very rich, allowing the user to interact with the application in many ways, leading to a large number of possible combinations of actions and possible flows to be executed. Web applications have multiple states, which are constantly changing according to user input. Testing this kind of applications became more complex, making the task of choosing what to test a challenge. Is it not feasible neither realistic to test everything, so prioritization is important to select a subset of test cases with maximum possible coverage in order to reduce the effort and costs of testing.

In addition, some testing approaches do not reflect actual usage of the application. It would be useful to build a test suite of regression test adapted to the application testing objectives, reflecting its real usage.

The main motivation of this work is to provide a way of generating automated GUI tests without effort, properly adapted to the application needs and reflecting real usage.

1.3 Problem and Research Question

This research work presents an approach that intends to generate test cases automatically from real usage of the Web application under test. MARTT - Mining Automated Regression Testing Tool - is the result of this work. This tool records user interactions, in a way similar as Web analytics tools do, and after, such interactions are analysed to generate test cases that may work as a regression test suite to be run every time an operating software system or its environment changes.

The approach proposed here is focused on already developed applications, built not only without automated tests but also, most of the times, without any type of documentation, what prevents from any kind of test generation from models, specifications or other documentation. Generating test cases from usage would allow these applications to have better quality control, even without formal models, encouraging the evolution, ensuring the previous working behaviour.

Using real usage data provides information about the user behaviour and allows test cases selection to be more real, ensuring that all tested flows were, at least once, performed by some user. This process avoids to spent resources and effort in testing impossible and unrealistic scenarios.

Furthermore, there is no need to previously define test scenarios since test cases will be based on the saved user interactions. The testers only have to define the criteria to select a set of test cases. This subset is selected according to different filters defined in our tool. The available filters allow having different types of test suites for the same application just collecting data once. The set of test cases can be more complete or less, with focus on elements, interaction or pages and ordered in different ways. This way, the tester can generate the test suite according to the testing purpose, matching the product needs, selecting a subset from the whole test sessions.

This approach would allow building a test suite for applications built without automated tests in an easier way, with less effort than to build the test suite from scratch. MARTT turns every user into a potential tester.

With this investigation, we aim to answer the following research questions:

- **RQ1. Are we able to generate an executable test suite from real usage data?**
- **RQ2. How much coverage of the application, regarding pages, can we get with all sessions?**
- **RQ3. How efficient is the reduction of test cases based on the collected sessions, regarding elements and pages coverage?**

The case study on a real Web application allows us to get concrete conclusions about this work.

1.4 Structure of the Dissertation

The next chapter (2) presents the state of the art about the automatic generation of GUI test cases, usage information and regression testing, especially focused on generation from usage information. Our proposed approach is presented in Chapter 3. We explain the process to collect data, the data analysis and the test case generation. Chapter 4 presents a case study over a real existing Web application. Threats to validity, conclusions and future work are in Chapter 5.

Chapter 2

State of the Art

The purpose of the following sections is to provide context about the state of the art regarding automated GUI test cases generation techniques, usage information and the techniques that generate test cases in regression testing context from usage data, going through the three main phases of the test cases generation process. Firstly, the first section describes Automated GUI Testing and the types of techniques to perform it (2.1). Then, the second session provides an introduction to Usage Information (2.2). Finally, the third section presents Regression Testing and the three main phases associated (2.3).

2.1 Automated GUI Testing

Nowadays, Graphical user interfaces (GUI) have a very important role in software, making the software friendlier for the users, allowing them to perform tasks in an easier and more intuitive way [34].

In this context, there are four main approaches that allow automating the generation of GUI test cases: *random testing*, *scripting*, *capture/replay* and *model-based testing*.

Each of these techniques can be chosen to be applied in a project according to different factors. The most suitable approach depends on, for instance, the main purpose and testing objectives, evaluation criteria, inputs and outputs, and criteria for stopping the test. It always depends on the testing that needs to be performed in the application and the objectives that testing aims to reach [29].

These techniques will be addressed in the following topics.

2.1.1 Random Testing

Random Testing is also called Monkey Testing. This type of testing allows performing random actions, either with the mouse or keyboard. There are three types of random testing. The first one, the *Dumb monkeys*, is not aware of which kind of inputs or outputs the systems allows to, neither of the state of the application under test, being its only goal to crash the application. The *Semismart* monkeys recognise bugs, while the *Smart monkeys* have some knowledge about the system, retrieving the data needed from a state table or model. In fact, Random Testing has high fault finding ability, sometimes even more than structured techniques, since the input coverage is very high and diverse. One advantage of automatic exploration and random testing is the possibility to explore less frequent paths that would not be tested manually. However, it usually tests unreal flow, not representing actual possible usage [35].

RVGT (Random Visual GUI Testing) script takes the most out of *VGT* (Visual GUI Testing). VGT is supported by tools that interact with the application through its GUI on a bitmap level, what means that the interaction is made against what is actually shown to the user. Using scripts based on scenarios and image recognition, the VGT simulates user actions. It can be used to automate random testing or exploratory testing actions. Automating GUI random testing, combining random testing and image recognition, RVGT is written using Sikuli, an open source VGT tool, built in Python, that allows interacting with any bitmap. The first part of the script provides a set of configuration variables, while the second part contains the GUI bitmap components [2].

Combining UI patterns and reverse engineering, Morgado et al. [38, 36] propose iMPAcT tool to test recurring behaviour represented by UI patterns in Android mobile applications using test patterns. This process is fully automatic, executing events chosen randomly. The order which events are executed may influence the testing results.

Some hybrid approaches are being developed in order to add randomly generated interactions to the GUI testing cases. In an experiment conducted using a mature open source application, it was concluded that "on average the added random interactions increased the number of visited application windows per test by 23.6% and code coverage by 12.9%. Running the enhanced tests revealed three new defects." [54].

Some approaches explore automatically the Web application using *Web crawlers* in order to generate test cases. *Web crawlers* are tools that systematically visit all pages of a website. However, since there is no human knowledge involved in knowing which test cases are needed to assert the application behaviour, generating proper assertions automatically is also a challenge. [14, 11, 6].

VeriWeb is a tool for automatically and systematically exploring a website, representing all possible execution paths followed by a user. This tool can also navigate through dynamic components as form submissions. VeriWeb is a tool that combines capture/replay capabilities and *Web crawler'* level of automation. Exploring paths implies three main tasks. The first one aims to exercise all possible paths a user might follow in the website. To do that, two different components are used: one to search all active objects (i.e objects with default actions associated) in all DOM of an HTML page and another one to explore the state spaces, controlling and observing the execution of all components. The second task is all about execution of the exploration selected actions by the Web Navigator. Web Navigator simulates user interaction with a browser. The last task focuses on error handling, dealing with navigation error and page errors and saving error scenarios in a file, which can be used for debugging purpose [6].

2.1.2 Capture and Replay

The Capture and Replay technique aims to record tester's interactions with the GUI, such as mouse motions, mouse clicks and keyboard inputs, replaying them afterwards when the system needs to be tested. The main goal is to be applied as regression testing, asserting expected UI behaviour and checking if the previously recorded behaviour is kept. Test Scripts are generated from the recorded actions in order to interact with the DOM when executing them. Despite being automatically generated, the tester still can change the scripts [20, 6, 42, 35].

Although this method eases the test script construction process, it still requires too much manual effort in defining test scenarios, recording them all, analyzing and, when it is needed, refactoring the generated scripts [20, 6, 42, 35].

This method is most useful at the end of the development testing, due to the relative lack of robustness of test scenarios, since recorded test scenarios became impossible to use very easily, with very small GUI changes. In this situation, maintaining test scripts will require a large effort that usually leads to discarding test cases. On the other hand, this method can also be used in early-stage prototypes since there are no bugs on implementation interrupting the capturing of test scripts [35].

Capture and Replay is one of the most popular GUI testing approaches, what leads to a big number of existent Capture and Replay tools in this field, such as *Selenium*[1], *Ranorex*[2] and *TestComplete*[3] between others [20, 6, 42, 35].

One of the most used and known capture/replay tools is *Selenium*. Selenium is an open-source set of different tools. Selenium IDE is a prototyping tool that allows to record actions and to build test scripts accordingly. As a Firefox and Chrome plugin, it is easier and quicker to use and to develop automated tests.

Ranorex is a full-featured automation framework that allows testing mobile, desktop and Web applications. It also provides a capture/replay tool, as well as support cross-browser testing and cross-platform mobile app testing. Furthermore, Ranorex supports image-based recording and object recording, offering a wide range of possibilities to the tester. Based on Xpath, Ranorex created its unique expression to identify UI elements: RanoreXPath.

There are some tools trying to improve on some fewer explored fields. For instance, *TestComplete* is using Artificial intelligence to recognise objects and controls and to improve its capture/replay method, through powerful object identification engine, allowing the user to build scripts with simpler and easier language, almost resumed to keywords. It also provides specific checkpoints that allow testers to check the application state while test cases are being executed.

2.1.3 Scripting

The scripting approach relies on writing test instructions as code, using one of several possible programming languages, such as *Ruby* or *Java*. To help writing the test scripts, there are several

[1]https://www.seleniumhq.org/
[2]https://www.ranorex.com/
[3]https://smartbear.com/product/testcomplete/overview/

APIs such as *Selenium*[4], *TestComplete*[5], *CasperJS*[6] or *Protractor*[7]. These object-oriented APIs allow interacting with the Web browser by making direct calls and checking the expected behaviour. Mainly, these APIs expect, in each test command, the object identifier to interact with, as well as the type of action to perform.

As referred before, *Selenium 2.0* is a cohesive object-oriented API to be used in automated testing, interacting with the Web browser, making direct calls to it, using each browser's native support for automation. This tool aims to test automatically Web applications and to verify the expected behaviour. Providing an easy API, Selenium allows either building, reading and maintaining test scripts in an easy way. Several different commands are provided in order to interact with the Web application and test it. In addition, Selenium Grid provides a solution for large-scale test suites, running test in parallel, in different environments, boosting the performance of the test suite.

CasperJS is a tool used for scripting and testing, which provides many functions to handle different interactions with the Web application. CasperJS uses selectors (CSS3 selectors or XPath expressions) to work with the DOM elements, allowing the tester to build and run several different scenarios on the browser.

Interacting with the application and performing actions as a user make, *Protractor* is an end-to-end test framework for Angular and AngularJS applications. This framework needs two files to run: a spec and a configuration file. Protractor, as the previously referred tools, interacts with HTML elements which locators are provided. Supported locators are CSS selector, id or name, as well as model (ng-model) and binding (element bound to a given variable). The two last types of locators are only supported on AngularJS applications.

Besides these tools, there are more available, following the same purpose. Maveryx[8], Katalon[9] and Froglogic[10] are examples of tools that provide APIs to build the test scripts and run on the application under test.

2.1.4 Model-based Testing

Model-based testing comprises three main key elements. The first is the model that contains information about the structure and behaviour of the application under test. It must be updated whenever there are changes so that the tests are also updated accordingly. Then, the tool that generates the infrastructure for the tests, including the output. These tools are fed with a test-generation algorithm, the criteria that define how to choose the set of test cases and which must be selected [11, 40].

[4]https://www.seleniumhq.org/
[5]https://smartbear.com/product/testcomplete/overview/
[6]http://casperjs.org/
[7]https://www.protractortest.or
[8]https://www.maveryx.com/automated-web-testing/
[9]https://www.katalon.com/
[10]https://www.froglogic.com/squish/

Model-based testing is very useful for systems that often change or that are still under evolution. Although test cases have to be regenerated and the model has to be updated, it is usually easier to maintain than create test cases from the beginning [40].

Model-based techniques are commonly composed of automated and non-automated steps to generate test cases. The level of automation affects the cost, time and effort of test cases generation. An automated approach means less time and effort, thus less cost [40].

Some model-based testing approaches for the user interface of Web applications have been developed. The approach proposed by Torsel [51] aims to infer a model creation and its maintenance. The model must reflect the application structure, its Web pages and their connections. Structural inspection of the model results in a dependency graph where there is a vertex of every edge in the application graph. With test case generation purpose, an algorithm selects paths to be executed from the graph. To transform test cases into actually runnable scripts, implementation details should be kept in an external UI mapping XML file. Test cases also need to be translated to the appropriate language, according to the test automation tool that will be used. To do so, Eclipse Xpand tool is used, transforming provided test cases into the chosen language [51].

Furthermore, path expressions can also be deducted from a website graph. As a starting point of the previously referred technique is *TestWeb*, a tool proposed to generate test cases from a set of path expressions deducted from a website graph. This model is previously computed by the *ReWeb* tool, also proposed on the same study along with *TestWeb*. In this context, a path expression is an algebraic representation of the paths in the graph. The test cases are generated from the path expressions, according to defined test criteria [45].

Deriving test cases from UML diagrams is a common process in Model-based Testing. Riebish et al. [46] propose a technique that transforms use cases (in templates form) into state diagrams and then, state diagrams into usage graphs from which usage model is derived, using an XML-based tool. Finally, test cases are generated from usage models. Besides the reduction in testing efforts, this technique also aims to support requirements engineering by applying use case diagrams.

Another approach was developed with the purpose to combine visual modelling and formal modelling notations. Combining the visual, based in UML, with the formal specification language, like Spec# [5], it is possible to convert a subset of UML in Spec#, reducing the effort of GUI modelling and its following translation. Firstly, UML diagrams must be modelled to test the usage, structure and behaviour of the GUI. After having the visual model, a formal model in Spec# will be derived from the UML model, which can be refined after, in order to become an executable model. Not only test cases can be generated by *Spec Explorer* from the Spec formal model, but also the degree of coverage of the UML behavioural diagrams can be analyzed. To finish the process, the tester should relate the user actions defined in the model with the GUI elements of the application under test. Finally, test cases can be executed [42].

In the technique proposed by Chen et al. [10], activity diagrams are used to represent the system behaviour while the elements of the diagram represent requirements attributes. The paths are then derived from the diagram and each test case takes a specific one with a specific set of data. A similar technique was proposed by Sanj et al. [16], specifying use cases using activity diagrams and generating test cases from it. Fernandez-Sanz et al. [16] also propose a specification-based approach to generate test cases from activity diagrams.

Moreover, some approaches focus on the representation of the application state to generate test cases. Usually, finite state machines are composed by a finite set of states, a finite set of inputs and a finite set of outputs. Moreover, they are also defined by a transition function, which receiving the defined input moves to the next state and produces the defined output. A finite state machine can be represented by a state diagram, a graph that represents states and state transitions by its vertices and edges, accordingly. The work of Bertolino et al. [7] and Gnesi et al. [18] propose generating test cases from UML state diagrams [26, 29].

In fact, Web applications are systems in which transitions occur from Web pages to Web pages, depending on the inputs, in this context represented mainly by user actions, and the current state, changing the state of the application to another one. There are a few testing techniques based on finite state machines since they suit the Web applications context, dealing with the states transitions in an efficient way. However, a Web application can have very large possible states resulting from the innumerable possible inputs and options. This can lead to the *state space explosion problem*. Finite state machines must be balanced in order to achieve an efficient testing coverage without being too small neither too wide, avoiding the problem of having too much possible states [29].

In addition, the work of Abdurazik et al. [1] generates test cases from collaboration diagrams, including both static and dynamic testing. The sequence path represented on the diagram must be executed at least by one generated test case.

Bringing AI to testing context, *PATHS* (Planning Assisted Tester for grapHical user interface Systems) is an approach for partially automating testing of GUIs that uses AI planning techniques to generate sequences of actions. The focus is on specify goals instead of sequences of actions. The process of test cases generation starts with the generation of an abstract model of the GUI under test, which is used to generate a set of operators, representing the user interface events. Preconditions and effects of the operators have to be defined by the tester. On the second phase, the tester defines the scenarios, specifying the initial and goal states. PATHS generates a set of test cases for each scenario, allowing the tester to change them. For the same goal, multiple and alternative plans are generated [32].

Furthermore, also generating test cases based on defined goals, there is an approach called *Pattern-based GUI testing*. A *UI Test Pattern* is defined in the form of: < Goal, V, A, C, P >. The goal variable is the name or the id of the test. The *V* parameter is a set of pairs with the possible input data to each variable, while the *A* is the sequence of actions to perform, representing how to execute the test. *C* is the set of the checks to perform during the testing run, which means the final purpose of the test (for instance, *"check if the title is shown"*). These parameters are set by the developer when implementing the model. The last variable, *P*, aims to represent when the tests can be executed and in which application's states they can run. This model-based testing approach aims to systematize and automate the GUI testing process, creating generic test strategies. Currently, UI Patterns are commonly used to solve some UI problems. For instance, the usual login appearance on most of the websites is mainly composed of two inputs (one for the email and another to the password) as well as a submit button. This solution promotes to reuse test cases, either within the same project or between other applications, reducing costs and effort [33, 35, 34].

PBGT supporting tool also uses UI Test Patterns, which are defined using *PARADIGM*, a domain specific language developed for this purpose, allowing to specify relations between patterns and to structure models in different levels of abstraction. PBGT process goes through six main steps. In the first one, tester drags and drops the elements (*Nodes*) of the PARADIGM language, connecting them with connectors (*Links*). After having the model, the tester needs to configure the input data and preconditions, as well as checks to perform during the testing run. In this step, it is possible to adapt common strategies to different applications. Then, fully automatically, test cases are generated, according to the configurations specified before. All paths, from the initial node to the end one, are generated. After that, considering that is needed to map the model UI test patterns to the UI patterns on the application, each pattern must be selected and related with the GUI controls manually. The test cases are executed next, executing all the testing actions. The process finishes with a report of the testing results which may be analyzed by the tester [33].

On the other side, there is the opposite approach: produce formal models extracting information from the GUI under analysis by a reverse engineering technique. *ReGUI* is a tool that allows generating different types of models to represent different types of information. It is possible to generate graphical representations to check the visual aspect of the GUI, text models to use in model-based GUI testing and a Symbolic Model Verification represented by computation tree logic. This tool aims to generate visual and formal models with less effort [37].

A different approach is presented in [14]. In this case, the authors infer a model from existing test cases and then feed a crawler with those tests in order to generate new test cases. Additionally, the test suite can be extended for uncovered or unchecked parts of the Web application. This approach, named *Testilizer*, can use provided assertions to generate new assertions on the new generated test cases. The state exploration component is built on top of *Crawljax*[11].

In order to reduce the effort associated with the test cases generation, avoiding to build formal models from scratch, and to create test cases based on real information, there are some approaches that generate models from usage information. Next, usage information will be introduced.

2.2 Usage Information

There are two main methods to collect usage data from websites: one is by Web server log file method and the other one is page tagging method [24].

The first one handles tracking files stored on a Web host server. These files can save information about access, like time, place or IP address, as well as which pages were visited [24]. Information gather from server side often have some trouble dealing with user actions identification. It is difficult to identify the clickstream as well as the path followed by the user, leading to some lacks on the tracking. So that, extracting the exact path that a user followed can be a difficult task [23].

[11] http://crawljax.com/

On the other hand, page tagging methods allow retrieving data by JavaScript. This method does not rely on log files. Furthermore, even when pages are cached, the user's behaviour is still recorded, what does not happen on Web server log file method, since there is no request to the server [24].

Web server log file method is not able to track events, only downloads. In contrast, page tagging allows tracking events, downloads and page views. However, only focusing on downloads from the server is not reliable, because they may not be registered, for instance, when the connection is very slow, the tracking may be lost. Single page applications can also be a limitation to log file method because only API calls will be recorded, instead of all elements interacted on the client side. JavaScript methods are more efficient also at visitor tracking because log file method has some trouble to calculate returning visitors or number of visits [24].

However, on the security context, Web server log file method stays ahead because it does not need additional files to track data. In contrast, page tagging method could be a security threat if someone accesses the Web server and inject any code to the website, collecting sensible data [24].

Due to privacy rules, such as the *General Data Protection Regulation*, usage data must be processed to avoid retrieving any personal information.

Usage information has been used with multiple purposes: it may be used to understand the customer in a commercial point of view, which is the main purpose of Web analytics tools. These tools, such as *Google Analytics* [12], *Clicky* [13], *Open Web Analytics* [14], between others, provide knowledge to know customers' preferences in order to personalize the information that is shown to them.

Moreover, usage information is also important in other fields. For instance, in requirements context, *REQAnalytics*, a recommender systems, collects usage information of the software, being able to relate that information to each requirement. Furthermore, reports are generated in order to get recommendations for changes to improve the system. This tool uses a Web analytics tool, called *Open Web Analytics*, to help to retrieve usage information. This information is analyzed afterwards, resulting in the recommendations report. The process to generate recommendations reports is similar to the one used to generate regression testing: collecting, mining and generating.

In an attempt to deal better with input information, to represent real usage or to reduce the effort to build test cases, usage information has been applied to generate test cases, building test suites that can be executed as regression testing.

2.3 Regression Testing

Regression Testing is defined by ISTQB as "Testing of a previously tested component or system following modification to ensure that defects have not been introduced or have been uncovered in

[12]https://analytics.google.com/analytics/web/
[13]https://clicky.com/
[14]http://www.openwebanalytics.com/

unchanged areas of the software, as a result of the changes made" [8]. During software development, regression testing is what verifies that updates did not break previously working functionalities [20, 19].

Regression tests are important to ensure that software correctness is not affected by the changes performed. Any code change is likely to impact features that it is not expected to impact, so running regression testing is essential to make sure that no features are broken with the change neither the specifications previously defined. Regression tests can be run at any level: Unit, Integration, System, or Acceptance [44].

Since regression tests may be run several times, its automation gains more importance. However, regression test automation has some challenges, such as how to generate test cases, how to prioritize them and how to execute them.

The process of regression testing goes through three different steps: Test case Generation, Prioritization and Execution.

2.3.1 Test Case Generation based on Usage Information

The costs with regression testing are high, and because of that, it has been a concern to the newly developed approaches to generate test cases. In the previous section, different techniques to generate automated GUI test cases were addressed: random testing (2.1.1), capture and replay (2.1.2), scripting (2.1.3) and model-based testing (2.1.4). To avoid manual work to build test scripts, model-based approaches are common in the regression testing context. However, they are useful when there are models of the system under test. When such specifications do not exist and it is not trivial to construct them, it is necessary to resort to other techniques to derive test cases. In particular, there are approaches that gather information about Web applications usage and build test cases from that information. The main steps of the process are: gather usage information, analyze such information and generate test scripts.

Focusing on this work's context, this section will be specifically about test cases generation from usage information.

Generating test cases from usage information is more common for the Web. However, in [30], the authors mine the saved interactions, performed in Android apps, to derive execution scenarios using statistical language modelling, static and dynamic analysis. The data is collected from developers usage, which origin event logs that are saved. The logs and the source code are mined to build a vocabulary, which will be used to generate event sequences. These scenarios are validated executing the app in a real device.

In the Web field, there are more approaches. Some of those approaches use usage models to generate test cases [21, 52, 25, 11].

Usage models that were previously built based on actual usage scenarios and respective frequencies can help to select, execute and build test cases. Approaches in which test cases are generated from a graph, where the nodes represent the different application states and the arcs the transactions between them, help to prioritize test scenarios, focusing on more important features to test. This derivation from usage models are included in *Statistical Testing*, which aims to sample data when the data to potentially build test cases is too large [21, 52].

Amalfitano et al. [3] present an investigation about using execution traces for generating test cases. These execution traces can be obtained either by collecting usage data or by being automatically explored by a *Web crawler*. Each execution trace is converted in a test case. However, the generation of test cases depends on solving two problems. The first one is to get the pre-conditions of each test case, which can be solved by saving the application state before recording the execution trace. The second problem is regarding the testing oracle and how to assert the test success or failure. In the referred work, some solutions are presented, however, the authors define the testing failure by checking the occurrence of *Javascript* crashes.

One of the problems with collecting usage information is to define test input data since it should not be collected due to privacy laws. Many approaches still do not deal with input data, depending on a manual effort to provide it. Also depending on input data provided manually afterwards by the tester, in [43], test cases are generated from the most frequent interaction paths. To generate test cases from usage information retrieved from a *SaaS* under test, the process should go through four different steps. The process starts with *OWA* logs collection, followed by the calculation of the most frequent paths. After that, regression test scripts are generated, following a set of rules defined in a manual and automated process [43].

User session-based testing techniques retrieve, at the server site, URLs and name-value pairs that are the result of a sequence of interactions to generate test cases. A user session starts when a request from a new IP address is received by the server and ends when the user leaves the Web site or the session times out [13].

Kallepalli et al. [25] propose using log files from the server to build a usage model, using specific tools, for statistical Web testing and reliability analysis [25].

Based on session data, Elbaum et al. [13] proposed four different ways to generate test cases: the first one consists in replaying each individual request (formatted into an HTTP request to be sent); the second method replays a mixture of interactions from different sessions; the third aims to run sessions in parallel in order to have concurrent requests; the last one is to mix regular sessions to interactions that may cause some trouble, like backward and forward actions. However, the approach presented in [13] is only applied in a controlled environment.

It is shown in [13] that the percentage of faults detected increases as the number of user sessions increases. However, the time, effort and the cost associated with collecting the data, analyzing the data and generating the test cases also increase. For most of the applications, an exhaustive testing approach is infeasible, either technologically and economically. This leads to the need for test cases prioritization, with the purpose of getting a subset of test cases that can properly cover the application with a smaller number of test cases.

2.3.2 Test Case Prioritization

Testing user interfaces can be a challenge due to the number of possible combinations of actions that can be executed on the GUI. Usually, a GUI action can lead to multiple kinds of results, depending on the state in which the application is. Even small GUIs have several states and transactions, which means higher complexity on testing. If you exercise everything, you lose time, if you do not exercise enough, failures may not be detected. So, prioritization is important to select a subset of test cases with maximum possible coverage in order to reduce the effort and costs of testing [20, 32, 4, 19].

Test cases can be prioritized using coverage-based techniques, choosing the subset of test cases based on the code coverage they provide. These techniques can prioritize test cases based on statements, branch or functions covered [48].

On the other side, following a *black-box* approach and not looking into the code, test cases can be prioritized based on requirements: customer-assigned priority, based on the importance the customer gives to each requirement; requirements volatility, based on how much times a requirement has been changed during the development cycle; implementation complexity, based on how complex the development team perceives the implementation of the requirement to be and fault proneness, choosing the requirements that have been failing more [48].

Additionally, another technique of prioritization is observation-based testing, which aims to filter, from a large set of test cases, the ones that are most likely to match requirements and test needs. Filtering the test data allows selecting a smaller set of test cases which includes, for instance, interesting events or elements in the context of the application under test. This approach was already studied by Dickinson [12], clustering the population and sampling those clusters, in an attempt to find failures in a more efficient way. The filtering can also be used to produce execution profiles. It is also common to use a multivariate visualization technique that allows the tester to visualize the distribution of the data, understanding features that can be important to build the test cases accordingly [12, 27, 49].

Elbaum et al [13] propose two approaches to get a test suit size reduction. The first one is based on the technique presented by Harold et al. [22], which aims to get a representative set of test cases, removing redundant test cases and still achieving the intended testing coverage. In [13], this heuristic is applied to the test suite generated from usage data, reaching big reductions and keeping the coverage as intending or when it is necessary, compromising it by small margins.

The second approach presented in the study from Elbaum et al. [13] is based on clustering analysis, grouping similar test cases in clusters. Clustering methods are focused in grouping by similar properties, like similar browsing patterns or Web pages with semantically related content.

As well as clustering, also statistical analysis, association rules and sequential pattern analysis help to discover usage patterns that can be used to select a subset of test cases. Statistical Techniques are the most common ones, being able to get, for instance, the most frequently accessed pages, average visits, viewing time and average navigational path length. Association rule mining is used to find related pages that are most often accessed together in the same single session, showing that a relationship between items exists. This kind of rules states that in a transaction, if item 1 occurs, the probability of item 2 also being in the transaction is higher than if item 1 did not occur. Sequential pattern analysis aims to find relevant patterns between data sequences [31, 23, 41].

In [55] an attempt to reduce test cases based on URL similarities is presented. Firstly, URLs are matched between sessions, trying to find whether a URL is the prefix of another URL. Then, if a URL is contained in another one, it is removed from the test cases since its behaviour is already being tested by the URL from which it is a prefix. It is important to have in mind that this technique is applied to session-based testing, in which saved URLs are the request made in the sessions. To evaluate the similarity between URLs, parameters are also compared.

2.3.3 Test Case Execution

Regarding GUI Testing, there are three generations of tools: coordinate–, element– and image-based. Coordinate generation is less stable because it expects to get the same element on the same coordinates of the screen. With the increasing variety of devices and screens, this technique has become less useful [20].

To replace it, an element-based technique is used, aiming to use references to the elements of the GUI. This structure-aware technique is more precise than, not only the referred first generation but also than a human tester. Using, for instance, ID, *XPath*, type or label, grouped or combined, this technique can interact and perform several actions on the application. Element-based tools are more robust to minor layout changes than the other generations. Therefore, who writes the test needs to know about the GUI structure and its identifiers, which can limit who can write element-based tests [20].

On the other hand, Web test cases may be fragile when using Web page DOM. Since they are used to identify GUI objects, any change at this level leads to broken test cases. Whereas DOM is usually very dynamic, element locators may be a challenge. DOM element locators should be specific enough to get only the required element(s) [28].

Regarding the third generation, no structural elements are used, just image-processing techniques. This generation of image-based tools are more robust to code changes than element-based tools and are more useful to test and check changes only on the GUI. This complete black-box technique allows us to test the screens without knowledge of the application, switching easily between contexts, without depending on requests or third parties [20].

Despite all the advantages, each generation has its target and its purpose, according to the application's needs.

2.4 Summary

It was presented in this chapter the investigation context to automated GUI test cases generation techniques and regression testing, with main focus on approaches that implement usage information to derive test cases. In the first section (2.1), four ways to generate automated GUI tests are presented - random testing, capture and replay, scripting and model-based testing. Then, in section 2.2, it was introduced how to collect usage information and some of its usages, since Web analytics tools to requirements analysis and the possibility to implement it in the context of regression testing, which is described next, in the section 2.3. Regression testing is introduced by passing through its main three phases - test case generation (only presenting the techniques focused on usage information), test case prioritization and test case execution.

Finally, the approach presented in this work is different from the previously presented solutions and, as far as it was researched, there is no known equal approach. Although there are some approaches based on collecting usage data, MARTT collects data on the client side, making a selection of what is important to retrieve. After collecting usage data, we aim to provide a set of several different filters to select a subset of test cases. Another feature of our work is that, although we do not capture the input literally, we transform the input information in order to have information to get a close value to run the test script. In addition, our approach is applied on a real website, collecting real data in normal usage by its users and analyzing the captured data against the website's structure.

Our solution aims to collect the sequence of interactions made by the user to reproduce those interactions later. When we think on an individual session, this process can have some similarities with the capture and replay method, considering it is captured when the user performs it and replayed later, when running the test cases. However, if we look at the whole data set and how tests are generated, MARTT fits better on model-based testing. In spite of not having a formal model from which the test cases are generated, our usage data set can be considered as a model since we analyze that data and select the most important subsets with the purpose of generating test cases from that. This makes us believe that our approach can be included in model-based techniques.

Chapter 3

Proposed Approach

MARTT proposes an approach to generate test cases automatically from real usage of the Web application under test.

In this chapter, it is explained deeper how this work was implemented. The explanation is composed of four sections. In the Data Collection Section (3.2.1), the process of information capture is explained, as well as the properties of each interaction. The focus of the following Section (3.2.2) is the analysis process and the application built to analyse and to select the test cases. Then, it is explained how to generate a test script using the *Selenium* framework (3.2.3). To conclude, the technologies used in all development process are described (3.2.4).

3.1 Proposed Solution

The proposed solution is mainly composed of three phases.

In the first one, data regarding the interactions performed over the Web application under test is collected and then saved on a graph database: *Neo4j*[1]. In the second phase, saved data is analysed, according to the available filters. In the end, a JSON file is generated with the information about the sessions that match the defined criteria which is selected by the tester. In the third, and last phase, test scripts are generated from the JSON files.

These phases will be deeper explained next.

3.2 Implementation

3.2.1 Data collection

To retrieve user interactions from the website, a JavaScript file (*tracking.js*) must be included in the pages that will be under test. This file is responsible not only for collecting the data that is needed to replay the interaction later on but also to save it. In order to save the interaction, the file makes requests to the API responsible for connecting to the database.

[1]https://neo4j.com/

In terms of components, the process of data collection depends on the tracking file and capture data API which is represented by the numbers one and two, accordingly, in Figure 3.1.

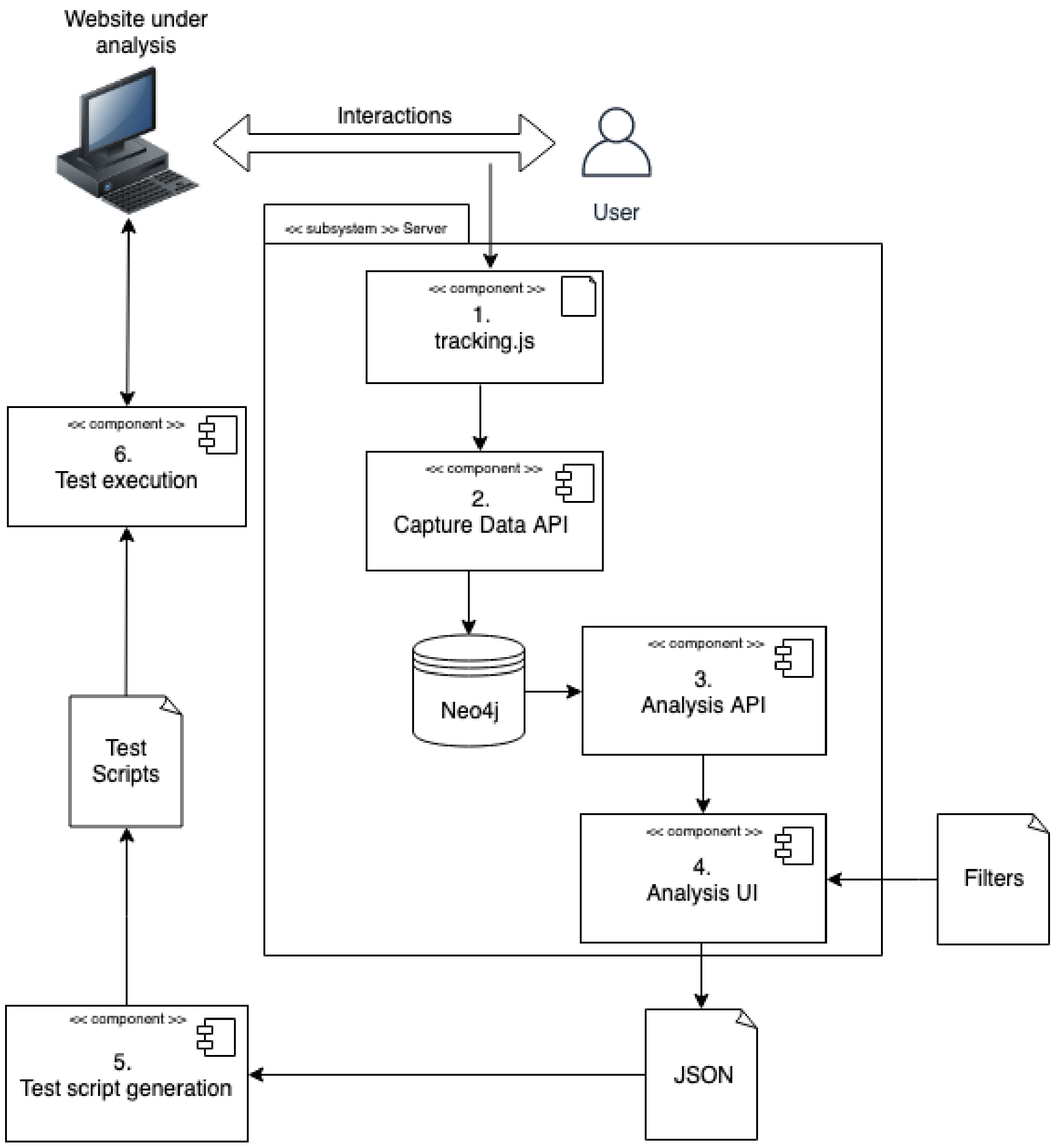


Figure 3.1: Components diagram of the proposed solution.

By including the JavaScript file in every page of the application, an event listener starts listening to events across the page — using the JavaScript function `addEventListener`. This listener will be triggered whenever the user interacts with the page, and consequently, an event is dispatched.

The events that are caught by the data capture script are: *click*, *double-click*, *drag and drop*, *key pressed* and *paste* events. Every time an event of these types is detected, the script file will retrieve from the page the information needed to playback the interaction .

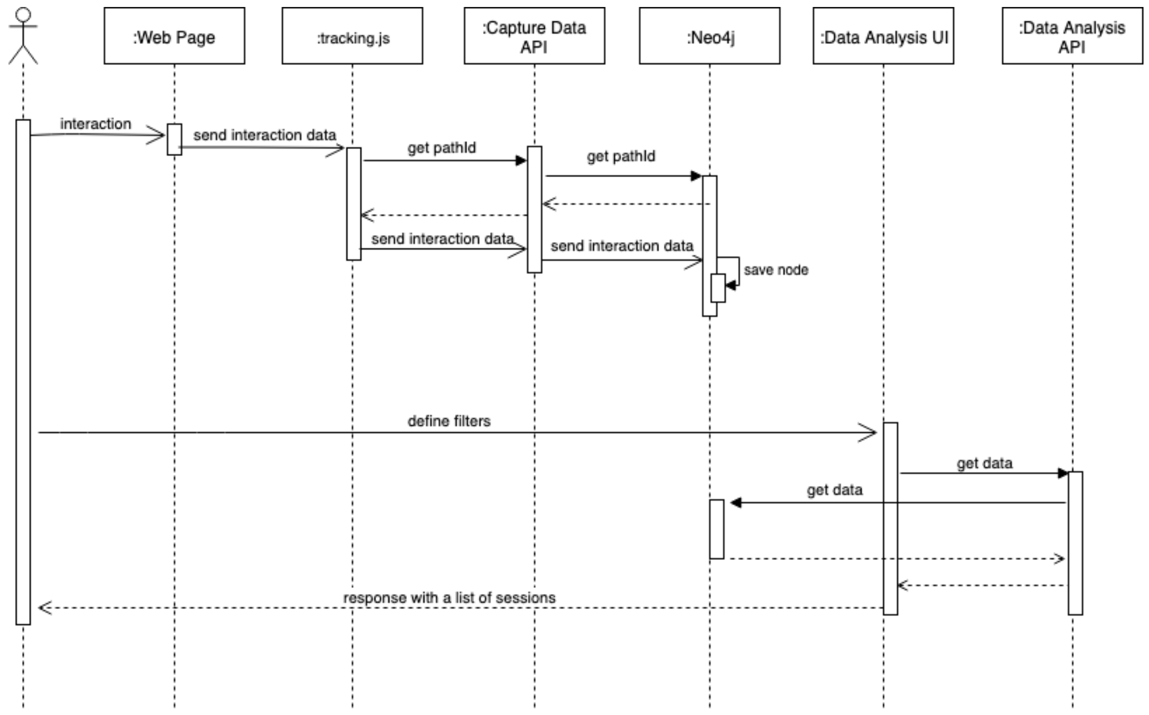


Figure 3.2: Sequence diagram of the proposed solution.

Each node on the database represents an interaction, as shown in Figure 3.3, structured as it is shown in Listing 3.1. It is possible to connect each node with other nodes using a relationship connection. If the action is the first of the session, it is saved as a simple node and without any relationship. On the other hand, if the interaction is not the first one, it is saved with a relationship with the previous interaction. This process is represented in the sequence diagram in Figure 3.2.

```
"properties": {
  "path": "id(\"search\")/input[@class=\"text\"]",
  "session": "dfdf5a5d-98a4-d90d-334d-094fb7180d80",
  "actionId": 3,
  "action": "input",
  "pathId": 5,
  "elementPos": 2,
  "value": [
    "char",
    "char",
    "char",
    "Enter"
  ],
  "url": "http://www.ipvc.pt/"
}
```

Listing 3.1: JSON structure of a node’s properties.

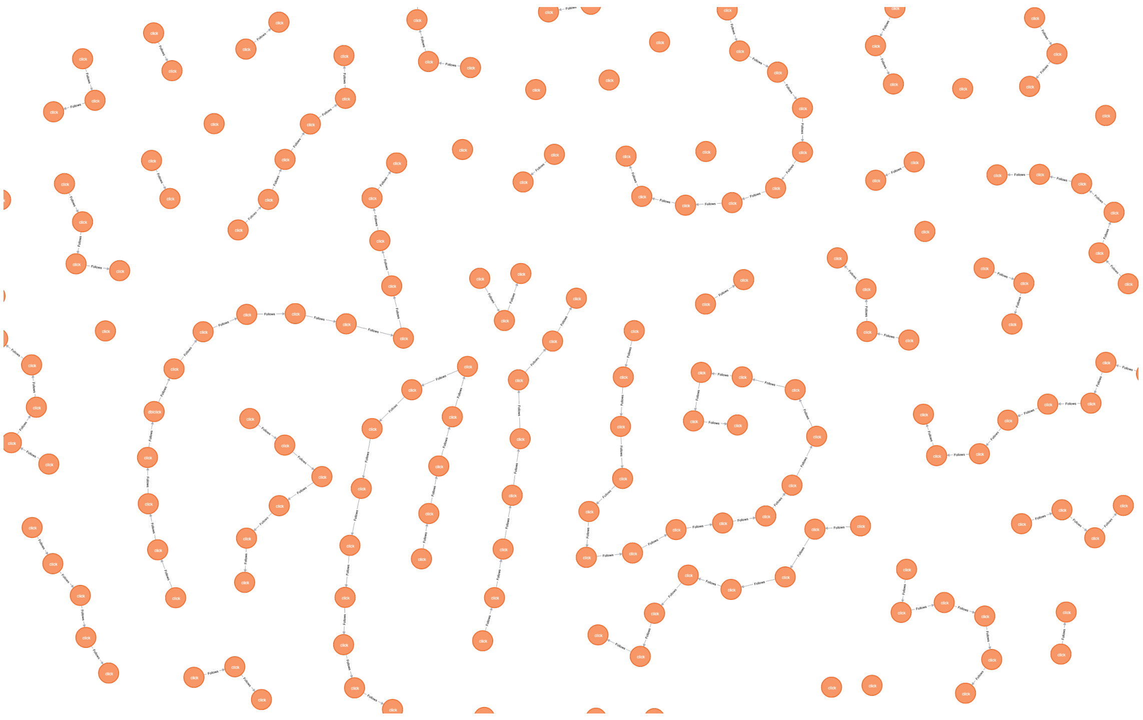


Figure 3.3: Snapshot of the Neo4j browser.

3.2.1.1 Session Identifier

In this work context, it is important to keep the data related by user sessions, to have a proper flow representing real linked interactions. This will be useful knowledge about the data for the analysis phase. To do so, session identifiers are used.

The goal is to connect interactions, firstly by *sessionID*. In order to be possible to group saved interactions by session, each user session (i.e. the tab where the website is running, since the moment user opens it until it is closed) is identified by a unique identifier composed of 32 random characters (either letters or numbers) which is generated every time a new tab is open.

3.2.1.2 Element Position

The session ID is enough to group the interactions. However, in order to reproduce them according to the same flow/sequence, the interactions have to be organized in the same order they were performed by the users.

To do so, another property is saved in each element that indicates the position of the element (`elementPos`) in the respective sequence. This variable is saved locally, in *session storage*, and incremented every time a new interaction is recorded. When a new session is created, with a respective new session ID, the position restarts, starting from 1.

This way, it's always possible to know where an interaction is in the path, as well as the previous and following interactions.

3.2.1.3 DOM Element

To replay each interaction, it is essential to have a way to identify precisely the element which the user interacted with. The two main discussed options were *CSS selector* and *XPath* [2].

CSS selectors are not much precise and may not retrieve the right element since some selectors, such classes, are not unique and can be repeated between elements. *XPath* stands for *XML Path Language* and allows to navigate through the structure of an XML document, representing the position of an element in the DOM (Document Object Model) using a path notation that embodies not only the structure tree but also the ID or class, if there is any. *XPath* is not only more precise than CSS selectors, but it also is independent of the properties added to the elements (such as IDs or classes). The work presented here is focused on already developed applications, making very risky to trust in CSS selectors, because the elements can have no specific selector associated. Based on that, a function was implemented in the *tracking.js* file that extracts the *XPath* from the interacted element. The *XPath* structure is similar to the following one [Listing 3.2].

```
id("node-4747")/div[@class="node-inner"]/div[@class="content"]/div[@class="field-
    type-text"]/p[2]
```

Listing 3.2: XPath example.

3.2.1.4 URL

The URL of the page in which the interaction is made is saved as a property of the interaction. This property ensures that two equal actions in different pages are not misunderstood and interpreted as the same actions since *XPath* is unique only in the respective page structure. Before being saved, the URL is formatted to remove parameters that are being sent through the URL in GET requests.

3.2.1.5 Action Types

The previous referred events, *click*, *double-click*, *drag and drop*, *key pressed* and *paste* events will trigger the process of retrieving information and saving it on the database. Associated with mouse events, three types of interactions are expected: *click*, *double click* and *drag and drop*. All action types are saved with a numeric ID beyond the string identifier, to be easy to analyse the data later if there is need to apply an algorithm that only accepts numerical values.

The *key pressed* event is triggered not only by all character keys but also by the *delete*, *tab*, *backspace* and *space* keys. The type of action that represents *key pressed* and *paste* events is the `input` action. This action has a complementary part with additional information that is better explained next.

[2]https://developer.mozilla.org/en-US/docs/Web/XPath

3.2.1.6 Value

According to each action type, an optional parameter is saved along with the action type identifier. In the *drag and drop* action, the *XPath* of the target element is saved in `Value` parameter, when the element with which the user is interacting with is placed at the destination of the drag and drop interaction.

In the *Input* action, despite each key triggering the event, the input data is saved as an array of keys to avoid saving a node for each key. To avoid saving sensitive and personal data, the input data is not saved literally. It is converted to *"char"* or *"string"* according to the length input. Moreover, when the selected input's type is a password, no input information is retrieved, making sure that no user information is disclosed.

When the input is made a character by character, the saved array may look like `[char, char, char]`. However, if the information is written all at once, like in a paste event, it is represented by a complete string and it may be saved like `[string]`.

Click and Double Click actions do not need to save any additional value to the action itself.

3.2.1.7 Path Id

Each set `[xPath, actionType, url]` is represented by a unique ID so that it can be used later on data analysis. As a result, each `pathId` represents a unique interaction with an element in a specific page.

So that, before saving a record, it is checked whether a specific set already exists or not in the database: if so, the previous ID is used; otherwise, a new ID is created, increasing the highest existent ID by one. The API should provide the `pathId` of the interaction to be possible to save the interaction performed by the user.

3.2.2 Data extract analysis

Using all captured data is neither feasible nor efficient. All the sessions that are important to test in the context of the application need to be retrieved. In the analysis process, the tester should define some constraints to result in a set of sessions that fits the tester needs as well as the testing purpose. As Figure 3.1 shows, filters defined by the tester are also an input of the Analysis UI, which allows to select and retrieve the subset of sessions. The process can also be seen in the sequence diagram in Figure 3.2.

3.2.2.1 Analysis API

To access the database and to get the requested data, it was developed an API. This API runs the *Cypher* queries, the graph query language native from Neo4j, on the database. In some endpoints, the API receives the filter criteria, reaching a set of sessions that respects the provided filter parameters. In other cases, all sessions are retrieved and selected afterwards on the front-end side according to the defined criteria.

3.2.2.2 Analysis Application

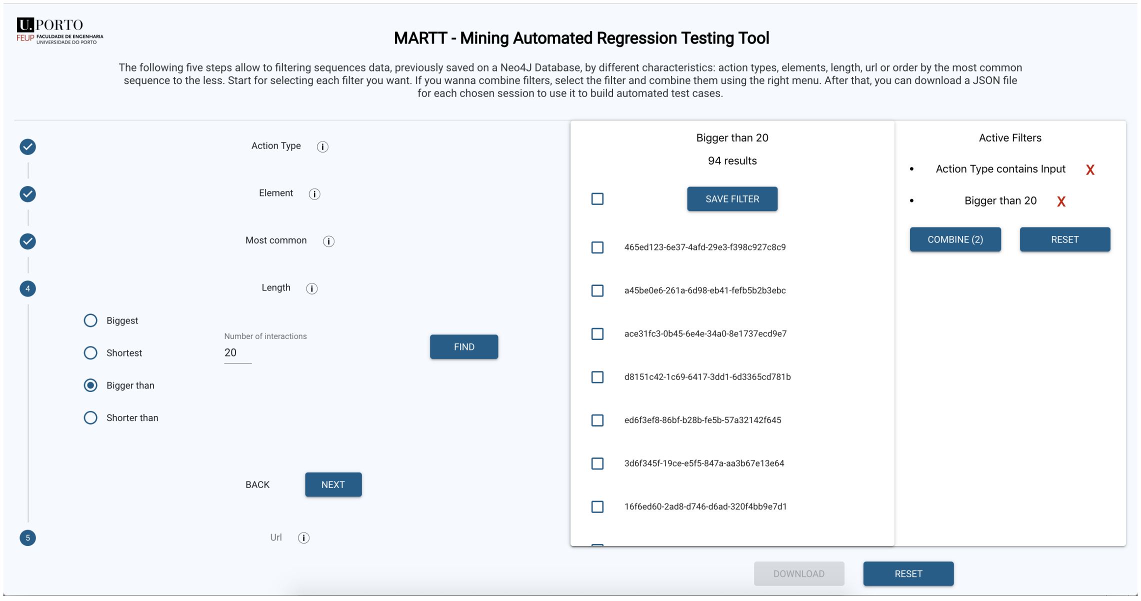
To make the analysis process more intuitive and visual, a UI application was developed as represented in Figure 3.4. More screenshots can be found in Appendix A.

This application allows the user/tester to make requests to the Analysis API, also developed in this work's context and previously explained, retrieving sessions data to be filtered and, afterwards, to generate the test cases.

The user can navigate through the filters panel, on the left, where the constraints can be defined. There are five main sections of filters: *Action type*, *Element*, *Most common sessions*, *Length* and *URL*. Then, in the middle of the screen, the sessions that correspond to the defined filter are shown.

If the result is interesting for the tester, the filter can be selected to have its result combined with other filters later one. When all filters are selected, the tester can combine them all and get the result that matches all the constraints. Then, it is possible to select the sessions to be downloaded in JSON format, as it is represented in the Components diagram in Figure 3.1, linking the Analysis UI to Test Script Generation. Sessions are downloaded in separated files. An example of a JSON file can be found in Appendix B.

This flow is represented in the sequence diagram, in Figure 3.2, as the process after saving the data. The process starts with defining the filter, followed by getting the data to result in a set of sessions that match the previously defined constraints. This process depends on the Analysis API and Analysis UI components, represented in the diagram components in Fig.3.1.


U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MARTT - Mining Automated Regression Testing Tool

The following five steps allow to filtering sequences data, previously saved on a Neo4J Database, by different characteristics: action types, elements, length, url or order by the most common sequence to the less. Start for selecting each filter you want. If you wanna combine filters, select the filter and combine them using the right menu. After that, you can download a JSON file for each chosen session to use it to build automated test cases.

Action Type
Element
Most common
Length

( ) Biggest
( ) Shortest
[✓ Bigger than]
( ) Shorter than

Number of interactions: 20

Url

Bigger than 20

94 results

☐ 465ed123-6e37-4afd-29e3-f398c927c8c9
☐ a45be0e6-261a-6d98-eb41-fefb5b2b3ebc
☐ ace31fc3-0b45-6e4e-34a0-8e1737ecd9e7
☐ d8151c42-1c69-6417-3dd1-6d3365cd781b
☐ ed6f3ef8-86bf-b28b-fe5b-57a32142f645
☐ 3d6f345f-19ce-e5f5-847a-aa3b67e13e64
☐ 16f6ed60-2ad8-d746-d6ad-320f4bb9e7d1

Active Filters

- Action Type contains Input
- Bigger than 20

Figure 3.4: UI application to analyse the previously collected data.

The filters available in the UI application are summarized next and better explained in the following paragraphs. The values used as criteria are provided by the tester on the UI interface. All the filters can be combined according to the tester preference in order to get the most suitable set of test cases.

1. Filter sessions by Action Types
    - More than X different action types
    - Less than X different action types
    - Equal to X different action types
    - Include the action type:
        - *Click*
        - *Double Click*
        - *Drag and Drop*
        - *Input*
2. Filter sessions by Element presence
    - Contain the *XPath* provided
    - Begin in the *XPath* provided
    - End in the *XPath* provided
    - Order by diverse *XPath* presence
3. Filter sessions by Length
    - The biggest session
    - The shortest session
    - Bigger than X interactions
    - Shorter than X interactions
4. Filter sessions by URL
    - Contain the URL provided
    - Order by diverse URL presence
5. Order by the most common session to the less common

3.2.2.3 Filter sessions by action types

The analysis application allows filtering sessions by the number of different types of interactions. It is possible to select the sessions that have more than a defined number of interactions, less than the provided threshold, as well as equal to it.

Moreover, the tester can define one type of interaction that the session should have, in case of needing to test a specific kind of interaction. This filter allows focusing the test on the type of interactions instead of the elements or pages. To have such data, a request to the `/actiontype` endpoint of the analysis API is needed (Listing 3.3), which retrieves all sessions and the number of different types of interactions each one has. The selection of the sessions that match the defined limit is made on the front-end side.

```
{
  "records": [
        {
            "keys": [
                "sessionId",
                "size(collect(DISTINCT n.action))",
                "collect(DISTINCT n.action)"
            ],
            "length": 3,
            "_fields": [
                "20099fd4-04b3-cbc5-b8ec-875842c830a7",
                {
                    "low": 2,
                    "high": 0
                },
                [
                    "click",
                    "dblclick"
                ]
            ],
            "_fieldLookup": {
                "sessionId": 0,
                "size(collect(DISTINCT n.action))": 1,
                "collect(DISTINCT n.action)": 2
            }
        },
    ]
}
```

Listing 3.3: one item of the /actiontype endpoint response.

3.2.2.4 Filter sessions by element presence

If the tester aims to test a specific component, it is possible to select sessions in which the element is present, providing the *XPath* of the element. Additionally, because the `/element` endpoint (Listing 3.4) provides not only the position of the element in the sequence but also if it is the last element or not, based in the sequence length, it can be filtered if the element is the first interaction or the last one. This type of analysis was inspired in the goal kind of approach, which main focus is in specifying goals instead of sequences of actions [32].

Furthermore, it is also possible to order the test cases by how much different new path IDs each test cover. In other words, the process starts with adding the test case that tests more different path IDs to the suite. All of the remaining test cases are analysed, one by one, to see which one adds more value to the suite. Each time one test case is added, the remaining are re-analysed, comparing the path IDs with the ones covered by the new test suite. When there is no new path ID to cover, no more sessions are added, avoiding to test sessions that do not add value to the suite.

```
{
  "records": [
        {
            "keys": [
                "sessionId",
                "count",
                "nodePosition",
                "lastNode"
            ],
            "length": 4,
            "_fields": [
                "cf48d0bf-4cc0-bc28-477c-85ec03c08d76",
                {
                    "low": 12,
                    "high": 0
                },
                6,
                false
            ],
            "_fieldLookup": {
                "sessionId": 0,
                "count": 1,
                "nodePosition": 2,
                "lastNode": 3
            }
        },
    ]
}
```

Listing 3.4: one item of the /element endpoint response.

3.2.2.5 Filter sessions by length

Based on the number of interactions, this filter allows choosing the longest or the shortest session as well as to define how many interactions the sessions should have, i.e., the number of nodes saved in each session. The `/allsessions` endpoint retrieves all sessions and the length of each one (Listing 3.5), ordered by the length in decreasing order.

The selection of the sessions that match the criteria is made on the front-end side, based on the length item. When the goal is to find the longest or the shortest session, it is picked the first or the last session, accordingly, of the order set of sessions.

This filter is useful to remove small sessions from the whole sessions set that do not add value to the test suite. Sometimes, saved sessions are composed of just one or two clicks. It is important to clean up the data, avoiding to waste effort testing useless sessions. Filtering by length allows managing coverage having fewer test cases with more performed actions.

```
{
  "records": [
        {
            "keys": [
                "sessionId",
                "count"
            ],
            "length": 2,
            "_fields": [
                "465ed123-6e37-4afd-29e3-f398c927c8c9",
                {
                    "low": 85,
                    "high": 0
                }
            ],
            "_fieldLookup": {
                "sessionId": 0,
                "count": 1
            }
        },
    ]
}
```

Listing 3.5: one item of the /allsessions endpoint response.

3.2.2.6 Filter session by URL

Ensuring that the test case interacts with a specific page can also be very useful. To do so, there is a filter that allows choosing a URL for which the sessions will be filtered. A POST request is made to the `/url` endpoint (Listing 3.6), sending the wanted URL as `url` parameter. The API returns every session that contains interactions made on that page.

In addition, it is possible to order the test cases depending on the value they add to the test suite, i.e. a test case is added to the suite when it is the one that adds more different URLs to the set of URLs. The first selected test case is the one which covers more different URLs. Then, the following picked test case is the one which covers more different pages, without the pages covered by the previous test case. The process is repeated successively, calculating the difference between the URLS covered by each session and the ones covered by the already selected test cases, until all different URLs are covered by the test suite. This structured selection process allows getting the same coverage that the whole set of sessions would achieve with only a subset. In addition, if there is a need to test only a part of the complete test suite, it is assured that selecting by order from the ordered test suite, the most complete test cases reduction is being selected. To get the data properly structured to this analysis, a request to `/urlsession` is needed. The response contains all sessions and the URLs each session includes (Listing 3.7). The algorithm used to analyse this information is explained in *Pseudocode* in Algorithm 1.

```
{
  "records": [{
            "keys": [
                "sessionId"
            ],
            "length": 1,
            "_fields": [
                "cf48d0bf-4cc0-bc28-477c-85ec03c08d76"
            ],
            "_fieldLookup": {
                "sessionId": 0
            }
        }]
}
```

Listing 3.6: one item of the /url endpoint response.

```
{
  "records": [{
        "keys": [
          "sessionId",
          "urlArray",
          "count"
        ],
        "length": 3,
        "_fields": [
          "ab0b2c3b-09d2-9d67-b4e9-21275f13ff6b",
          ["http://www.ipvc.pt/",
            "http://www.ipvc.pt/instituicao",
            "http://www.ipvc.pt/servicos",
            "http://www.ipvc.pt/recursos-humanos",
            "http://www.ipvc.pt/recursos-humanos-procedimentos-concursais"],
          {
            "low": 5,
            "high": 0
          }
        ],
        "_fieldLookup": {
          "sessionId": 0,
          "urlArray": 1,
          "count": 2
        }
  },
    ]}
```

Listing 3.7: one item of the /urlsession endpoint response.

```
Data: the response of the request to /urlsession endpoint (Listing 3.7) as a set of
      sessionId-URLs pairs
Result: a set of sessionIds ordered by the most diverse URLs tested
// the algorithm starts here;
1 $testedUrls \leftarrow []$;
2 $sessions \leftarrow []$;
  foreach session in data do
      if first session then
          Add $session.url$ to $testedUrls$ ;
          Add $session.id$ to $sessions$ ;
          removeSession() // remove session from the set of total sessions ;
      else
          while there is different URLs to add do
              foreach session in sessions to add do
                  getUrlsNotInUrlsToTest() // get how many URLs covered by the session
                    are not included in the ones already selected ;
              end
              orderSessionsByUrls() // order sessions by the one that tests more URLs not
                covered to the one that tests less ;
3             $session \leftarrow$ getFirstSession() // get first session: the one which tests more
                uncovered URLs ;
              Add $session.url$ to $testedUrls$ ;
              Add $session.id$ to $sessions$ ;
              removeSession() // remove session from the set of total sessions ;
          end
      end
  end
```

Algorithm 1: Algorithm implemented to order sessions by the URLs each one covers.

3.2.2.7 Order by the most common to less common sessions

One of the main goals of the analysis part of this work was to found which set of interactions, represented by sessions, were more common in all saved data. Firstly, it was needed to understand what *common sequence* could mean in this context. It was concluded that the most common sequence is the sequence that has a higher similarity degree between itself and all of the other sequences.

To get common sequences, since *Neo4j* does not provide any known method to get the most repeated sequence, it was needed to evaluate the similarity degree of each sequence pair. Two sequences are similar when they share common sets `[xPath, actionType, url]`, represented by the `pathId`.

In order to calculate the similarity between two sequences of interactions, we explored first the native algorithms provided by the Neo4j database.

Neo4j provides five algorithms in the *Neo4j* Graph Algorithms library to measure the similarity between two sets: *Jaccard* [3], *Cosine*[4], *Pearson*[5], *Euclidean*[6] and *Overlap*[7] Similarity.

Jaccard Algorithm measures similarities between sets, being the similarity equal to the size of the intersection divided by the size of the union of two sets. To measure overlap between two sets, as the name denotes, the *Overlap* algorithm was used, which defines the similarity degree as the size of the intersection of two sets, divided by the size of the smaller of the two sets.

However, the result of the *Jaccard* and *Overlap* algorithms is not influenced by the order that the elements are, only by the number of repeated elements across the sets. The order is very important in this context, once some interactions depend on the previous one. For example, if an element is only available when a button is clicked, the action in that element depends if the button was clicked or not. In view of the fact of these algorithms do not take into account the order of the sequence, they are not adequate to find common sequences in this context.

On the other side, *Cosine*, *Pearson* and *Euclidean* evaluate the order elements appear in the interactions set. *Cosine* similarity is the dot product of the two vectors divided by the product of the two vectors' lengths, which means the cosine of the angle between two n-dimensional vectors in an n-dimensional space. *Pearson* algorithm is the covariance of the two n-dimensional vectors divided by the product of their standard deviations. Last but not least, *Euclidean* distance measures the straight line distance between two points in n-dimensional space.

However, to evaluate two sequences similarity, these algorithms require sequences with the same length. To solve this problem, it would be necessary to pre-process the data and to fill in the sequence with null actions to have both sequences with the same length. The changes have to be made after getting the data since changing directly in the database would lead to a loss of characteristics, as the original length of the sequence. Having the data changed outside the database, it would no longer make sense to use *Neo4j* algorithms to analyse the data.

[3]https://neo4j.com/docs/graph-algorithms/current/algorithms/similarity-jaccard/
[4]https://neo4j.com/docs/graph-algorithms/current/algorithms/similarity-cosine/
[5]https://neo4j.com/docs/graph-algorithms/current/algorithms/similarity-pearson/
[6]https://neo4j.com/docs/graph-algorithms/current/algorithms/similarity-euclidean/
[7]https://neo4j.com/docs/graph-algorithms/current/algorithms/similarity-overlap/

Considering that none of those solutions was adapted to our case, we tried to find an approach that could work on the front-end side.

A request to `/path` endpoint (Listing 3.8) retrieves all sessions with the corresponding sequences of path Ids, representing all interactions of each session. The data allow comparing sessions to find the similarity degree between them.

The algorithm that was found as the more suitable for this context was the *Levenshtein* [8] algorithm, explained in *Pseudocode* in Algorithm 2 [9]. This algorithm works as a string metric for measuring the difference between two sequences of characters, getting the smallest number of editions (insertions, deletions or substitutions) needed to change one string into another. That was the type of similarity needed between two sequences. As a result, the algorithm was implemented to use an array of integers (a sequence represented by the `pathId` of each node) instead of a string.

The *Levenshtein* distance value is divided by the bigger sequence length, to attenuate the length effect on the value and to get a value between 0 and 1, the percentage of how much similar two sequences are. Two sequences with distance value equal to 1 are totally similar, while two sequences with distance value equal to 0 are completely different. The algorithm to compare all sessions and get the distance values between them is explained in *Pseudocode* in Algorithm 3.

After comparing each pair of sequences, the most common sequence is the one with the smaller average of the distance between itself and the other sequences, what means that the average number of transformations needed to change itself in any other sequence is the smallest one.

```
{
  "records": [
        {
            "keys": [
                "sessionId",
                "pathArray"
            ],
            "length": 2,
            "_fields": [
                "465ed123-6e37-4afd-29e3-f398c927c8c9",
                [4676, 2750, 467, 468, 7445, 7446, 7447, 7447]
            ],
            "_fieldLookup": {
                "sessionId": 0,
                "pathArray": 1
            }
        },
    ]
}
```

Listing 3.8: one item of the /path endpoint response.

[8]https://dzone.com/articles/the-levenshtein-algorithm-1

[9]https://people.cs.pitt.edu/ kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm

```
Data: sequence A (seqA) and sequence B (seqB)
Result: the value of the distance between the provided sequences
// the algorithm starts here;
1 $a \leftarrow$ length of seqA;
2 $b \leftarrow$ length of seqB;
  if a equals to 0 then
      return b and exit;
  end
  if b equals to 0 then
      return a and exit
  end
  for $i = 0$ to $a$ do
3     $matrix[i,0] \leftarrow i$
  end
  for $j = 0$ to $b$ do
4     $matrix[0,j] \leftarrow j$
  end
  for $i = 1$ to $a$ do
      for $j = 1$ to $b$ do
          if seqA[i] equals to seqB[j] then
5             $cost \leftarrow 0$
          else
6             $cost \leftarrow 1$
          end
7         $matrix[i][j] \leftarrow minimum($
          $d[i-1,j]+1$ // The cell immediately above plus 1
          $d[i,j-1]+1$ // The cell immediately to the left plus 1
          $d[i-1,j-1]+cost$ // The cell diagonally above and to the left plus the cost
          )
      end
  end
8 $distance \leftarrow matrix[a,b]$ //The distance is found in the position a,b
```

Algorithm 2: Levenshtein algorithm.

3.2.3 Test Cases Generation

Each session is represented by the set of nodes that are associated with it. To build test cases from the information of each session, each node is transformed in a test command.

Using test frameworks as Selenium[10], it is possible to write test commands from the JSON file, previously downloaded from the Analysis Application presented before, represented in Figure.3.4. The conversion of JSON files to test scripts is the process that allows having the actual tests to run in the Web application under test from which the data were retrieved. This process is represented by the components 5 and 6 in Figure 3.1.

[10]https://www.seleniumhq.org/

```
Data: the response of the request to /path endpoint (Listing 3.8) as a set of
      sessionId-pathArray pairs
Result: a set of sessionIds order by the most to the less common session
// the algorithm starts here;
1 $length \leftarrow$ number of sessions;
// create an empty matrix length x length to store distance values;
for $i = 0$ to $length$ do
2     $matrix[i,0] \leftarrow []$
end
for $j = 0$ to $length$ do
3     $matrix[0,j] \leftarrow []$
end
for $i = 0$ to $length$ do
    for $j = 0$ to $length$ do
        if session[i] is equal to session[j] then
4           $distance \leftarrow 0$ //equal sessions have no distance ;
        else
5           $levenshteinValue \leftarrow getLevenshteinValue(session[i], session[j])$ // the distance
              value between sessions is calculated with the Levenshtein algorithm presented
              in Algorithm 2 ;
6           $distance \leftarrow 1 - levenshteinValue/biggerLength$ // the distance value is divided
              by the length of the bigger session. Then it is subtracted to one to get a
              percentual value ;
7           $matrix[i][j] \leftarrow distance$;
8           $matrix[j][i] \leftarrow distance$ // the values are assigned for the inverse combination of
              the same sessions to avoid duplicated effort;
        end
    end
end
foreach row in data do
    $sumDistanceByRow()$ // for each session is summed every distance value regarding the
      other sessions ;
    $orderSessionsByTotalDistance()$ //sessions are order by the higher to the lower
      distance value ;
end
```

Algorithm 3: Algorithm implemented to find the most common sessions.

The properties regarding each node are used to build a test command for each interaction. In the following example (Listing 3.9), it is reproduced a click action. However, all the collected actions can be played by Selenium functions. Knowing the *XPath* of the element, the action can be executed exactly in the same element. The test commands can be performed in the same order that they were made, following the position of the element on the sequence (`elementPos`). In the input type of action, the value parameter provides some information about the input, useful to reproduce the action with the most similar information possible. However, since real inputs are not captured to keep the privacy of the user and to follow the GDPR (*General Data Protection Regulation*) rules, generating proper input to reproduce the test is still a challenge.

```
driver.findElement(webdriver.By.xpath("id(\"caixa-pesquisa\")/input[@class=\"
    search_texto\"]")).click();
```

Listing 3.9: example of a Selenium test command written in JavaScript.

3.2.4 Technologies

To store data, it is used a graph database: *Neo4j* [11], with native graph storage and processing. A graph database is more focused on data relationships because it treats the relationship as data (not like structure as relational databases do), allowing you to query data relationships in real-time using *Cypher*, *Neo4j*'s query language. In addition, Graph databases lead to a more flexible model, making changes to the model an easier task with less impact than in other kinds of databases [50]. Each record in the database is saved as a node, as it can be seen in Figure 3.3. This visual browser is interesting, allowing us, in this work context, to distinguish sessions and see the connections between their interactions.

In order to access the database, *Neo4j* provides an official driver to connect via an HTTP request. A *Node.js* application was created to implement the *Neo4j* driver. *Node.js* executes *JavaScript* [12] outside the browser, allowing to server-side scripting. In addition, it was used *Express* [13], a minimal and flexible *Node.js* Web application framework, which helps to create Web applications and APIs. In this environment, it was possible to implement the *Neo4j* driver and configure it to access the database. After that, using the *Express* framework, the endpoints were created: the ones needed to retrieve information from the database and the ones to save the usage data. Both applications, to collect data and to analyse it, were built using the same referred technologies.

Javascript is an interpreted programming language with object-oriented capabilities. It allows not only the user to interact with the Web page, but also the Web browser to be controlled and content to be changed. The official name of the language stands for *ECMAScript*. This was the

[11]https://neo4j.com/
[12]https://developer.mozilla.org/en-US/docs/Web/JavaScript
[13]https://expressjs.com/

chosen language to develop the script to save the data since it was needed that the script ran on the client-side, in a Web browser, to retrieved and process the information needed [17].

Furthermore, to build the Analysis application (Figure 3.4), which provides the filters to select test cases, it was used *React* [14]. *React* is a *Javascript* library which helps to create interactive UIs, managing the render of components in a very efficient way, just rendering the right components according to state changes. This library allows organizing the UI code into components, providing methods to easily pass data through them. In addition, it was used *Material UI* [15], a *React* UI framework. Implementing Google's Material Design, this framework provides styles, themes and native components to help to build a consistent application.

Git Hub [16], a Web-based hosting service using *Git*, was used for version control.

Both APIs and the analysis application were installed and, during this investigation, were running on a server provided by the Software Engineering Laboratory, in the Faculty of Engineering of the University of Porto. The process to install and access is explained in the Installation Manual in Appendix C.

[14] https://reactjs.org/
[15] https://material-ui.com/
[16] https://github.com/

Chapter 4

Validation and Results

This chapter is devoted to our case study. The results obtained by using MARTT are analysed and some conclusions are presented.

In the first section, Section 4.1, the case study is introduced. Then, in Section 4.2, it is widely presented how interactions are spread by the different pages, getting the most visited pages, used as a subset for further analysis. In the third section, the coverage is analysed, focusing on the elements of the page and presenting the coverage of the elements, either interactive or not, in a small sample. Then, in Section 4.4, it is studied how much sessions from the whole set are needed to reach maximum coverage, i.e the same coverage achieved executing all captured sessions. This analysis is made regarding Path IDs (4.4.1) and URLs (4.4.2). The last section, Section (4.5) sums up the most important results and compares the three approaches between them.

4.1 Case Study

MARTT was applied on the website of the *Polytechnic Institute of Viana do Castelo* [1]. It is a website used by everyone who wants to know more about the institute, as well as by students who want to access the pages of the schools that are part of the institute. The script to collect data was included only in the public pages of the institute, not including schools individual pages neither private access areas. That means that only the pages under the (`ipvc.pt/`) domain were analysed.

The data was collected from 8.30am on the 8th April 2019 to 12 am on the 19th of April 2019. In total, 18124 interactions were saved, which leads to 4900 different sessions.

4.2 Interactions

In a universe of 18124 interactions, 38.61% of the total interactions are performed only in the 1% most visited URLs, represented by 11 pages. This subset was selected from the most visited set, represented in Table 4.1. The first 11 most visited URLs of the 1111 different ones that were

[1]http://www.ipvc.pt/

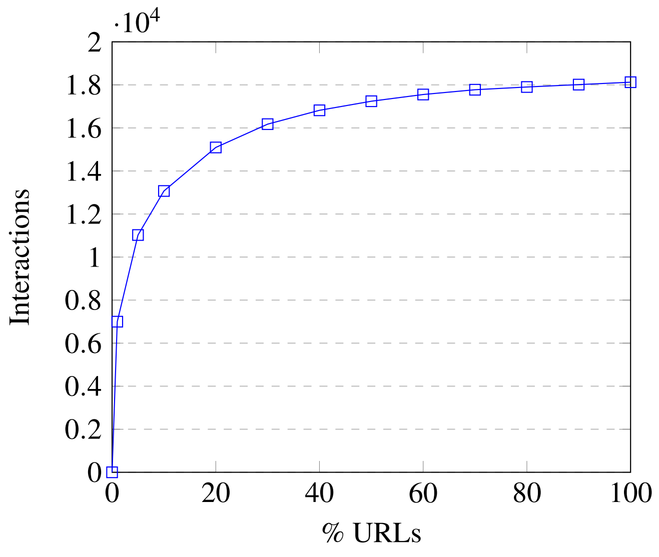


Figure 4.1: Relationship between the percentage of URLs order by most visited sessions and the number of collected interactions

saved during the process of collecting data are responsible for more than one-third of all retrieved interactions.

As might be expected, the homepage is the page with the higher total of collected interactions: 16.29% of the total interactions were performed on the first page (`http://www.ipvc.pt/`). As it can be observed in the Figure 4.1, at 40% of the most visited URLs, the number of interactions almost reaches the total of interactions, being 92% of the total captured interactions already performed. This means that the major part of the interactions is performed mainly in a small group of pages. However, the coverage of the pages will be deeper presented in the following section 4.4.2.

4.3 Elements

In this section, the focus is on the HTML elements presented on the page, either static or interactive. The purpose is to understand, in a small sample (1% most visited URLs), how the different type of elements are covered by the tests and the degree of their coverage, comparing the number of collected elements with the actual elements present in the pages.

Analyzing the number of elements that each page has and the number of elements that test cases would interact with, it can be concluded, by observing the data represented in Table 4.2, that the relationship between the number of tested elements and the URLs is not linear, at least not in the 1% most visited pages. In Figure 4.2, you can notice the irregularity of the interactive elements covered percentage. However, the higher percentage of tested elements, either interactive or not, matches with the most visited URL. The lowest percentage of interactive elements is found in the 9th URL. This page is one of the pages which have fewer interactive elements. Since it is a mainly informative page and not a much interactive page, users probably interact less with it, explaining why the percentage of coverage is lower than on the other pages. However, even in the other pages,

Order	URL	Interactions
1	http://www.ipvc.pt/	2952
2	http://www.ipvc.pt/m23-provas	630
3	http://www.ipvc.pt/servicos-web	547
4	http://www.ipvc.pt/instituicao	467
5	http://www.ipvc.pt/licenciaturas	429
6	http://www.ipvc.pt/mestrados	404
7	http://www.ipvc.pt/estudar-no-ipvc	376
8	http://www.ipvc.pt/ctesp	369
9	http://www.ipvc.pt/conselho-geral	354
10	http://www.ipvc.pt/mais-23anos	250
11	http://www.ipvc.pt/candidato	220
12	http://www.ipvc.pt/mestrado-enfermagem-medico-cirurgica	206
13	http://www.ipvc.pt/contacto	190
14	http://www.ipvc.pt/pos-graduacoes	162
15	http://www.ipvc.pt/pesquisa	161
16	http://www.ipvc.pt/recursos-humanos-procedimentos-concursais	153
17	http://www.ipvc.pt/maiores-23-candidaturas-2019-20-2-fase	144
18	http://www.ipvc.pt/calendario-escolar	144
19	http://www.ipvc.pt/eleicao-presidente-admissao-definitiva-candidaturas-2019	142
20	http://www.ipvc.pt/formacao-especializada	128
21	http://www.ipvc.pt/candidaturas-estudante-internacional-2019-20-2-fase	117
22	http://www.ipvc.pt/maiores-23-anos-resultados-seriacao	114
23	http://www.ipvc.pt/viver-no-ipvc	110
24	http://www.ipvc.pt/mestrado-gestao-organizacoes	102
Total		8871

Table 4.1: Most visited URLs

Order	URL	Total	Int.	Total Tested	Int. Tested	% Tested	% Int. Tested
1	http://www.ipvc.pt/ *	1302	140	259	114	19.89%	81.43%
2	http://www.ipvc.pt/m23-provas	941	171	79	64	8.39%	37.42%
3	http://www.ipvc.pt/servicos-web *	788	103	93	32	11.80%	31.07%
4	http://www.ipvc.pt/instituicao	804	106	68	42	8.45%	39.62%
5	http://www.ipvc.pt/licenciaturas *	910	125	69	38	7.58%	30.40%
6	http://www.ipvc.pt/mestrados *	956	134	68	35	7.11%	51.47%
7	http://www.ipvc.pt/estudar-no-ipvc	657	95	41	27	6.24%	28.42%
8	http://www.ipvc.pt/ctesp *	961	134	74	43	7.70%	32.09%
9	http://www.ipvc.pt/conselho-geral	857	99	73	16	8.51%	16.16%
10	http://www.ipvc.pt/mais-23anos	848	129	38	31	4.48%	24.03%
11	http://www.ipvc.pt/candidato	811	111	49	38	6.04%	34.23%
Total		9835	1347	911	480	8.74%	36.94%

Table 4.2: Elements tested on each URL of the 1% most visited subset

excepting the homepage, the coverage does not reach very high values. The interactive elements that pages have are mainly links to other pages, so the users likely interact with only a few of them, since the purpose of most of the pages is to be informative and static.

From the total number of different elements, i.e different *XPaths*, it was selected the interactive elements: *anchors*, *buttons* and *inputs*. The data presented in the Table 4.2 shows, for the 11 most visited URLs, the number of total elements, the selected interactive elements, the total and the interactive elements collected covered by the tests, the percentage of elements and interactive elements tested for each URL. It is important to notice that all of these pages have three hidden inputs that prevent, from the beginning, to achieve total coverage.

After retrieving all the elements, we noticed that some pages had more elements tested than exist on the page. That led to a deeper investigation that revealed that some elements of the `ipvc.pt` have dynamic classes that are composed by a random ID generated each time the page is loaded. It is usually a development decision when a list is rendered and there is a need for different keys in each item. This means that the *XPath* of these elements will be changing every time, leading to different *XPaths* and consequently interpreted as different elements. To solve this problem, the data was pre-processed before being analysed to remove that random ID. URLs in which this process occurred are marked in the Table 4.2 with an asterisk.

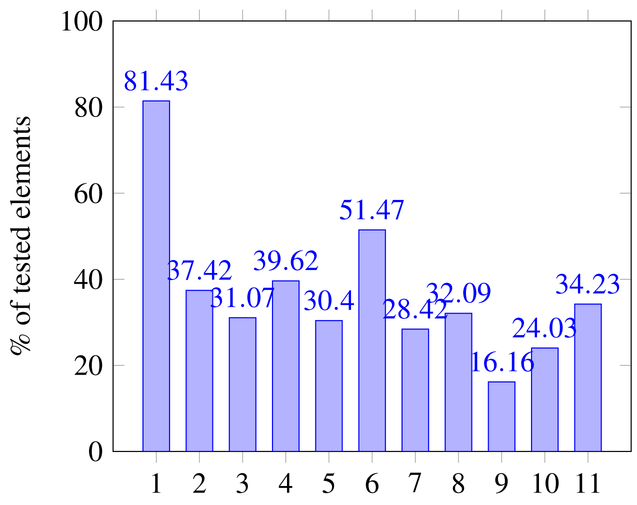


Figure 4.2: Relationship between the 1% most visited URLs subset and the percentage of interactive tested elements

4.4 Test Suite Reduction

Testing all sessions is not feasible. So, prioritization is important to select a subset of test cases with maximum possible coverage in order to reduce the effort and costs of testing. With test suite reduction we can achieve the same coverage as when executing all sessions. In this section it will be presented three different ways of selecting test cases: selecting test cases based on most common sessions, selecting test cases based on how diverse test sessions are and selecting test cases randomly. The three methods will be applied and analysed regarding Path IDs and URLs.

4.4.1 Path ID

The next proposal present here is still focused on the elements, however, instead of focus only on the XPath of the elements, it is focused on Path ID, in order to represent the whole action. The test cases were selected using the Analysis App, presented in the previous chapter, and the coverage of the collected sessions was analysed by selecting the tests following three different approaches: ordered by most common sessions, ordered by the sessions that test more diverse Path IDs and *Random approach*. In this section, the value used as a reference to analyse the coverage will be the number of total different collected Path IDs: 8333.

4.4.1.1 Order by Most Common Sessions

Ordering the test cases by the most common to the less common session, it can be concluded that the total coverage of the collected Path IDs can only be achieved when running the 4900 test cases. As it can be seen in Figure 4.3 and in Figure 4.4, the most common sessions test a very small number of Path IDs. To achieve 50% of coverage, it would need to run at least 60% of the test cases ordered by this method. This happens due to the short length of the most common sessions.

To improve the coverage results, the test cases were not only ordered by most common sessions but also filtered by length, getting different results. Choosing only sessions with more than 5 nodes, it is possible to achieve 29.27% of coverage of the Path IDs with 490 tests (10% of the previously analysed test suite), however, without reaching total coverage. The total selected tests with more than 5 interactions - 1090 test cases - would cover 70.91% of the total collected Path IDs. If the sessions are filtered by more than 10 nodes by session, the maximum coverage is 46.70% with 380 test cases, even fewer test cases than the 490 considered before. Although this combination of filters does not lead to total coverage, it leads to higher coverage with fewer test cases. So, it may fit testing needs, depending on the coverage goal and what is intended by the testing team or product goals.

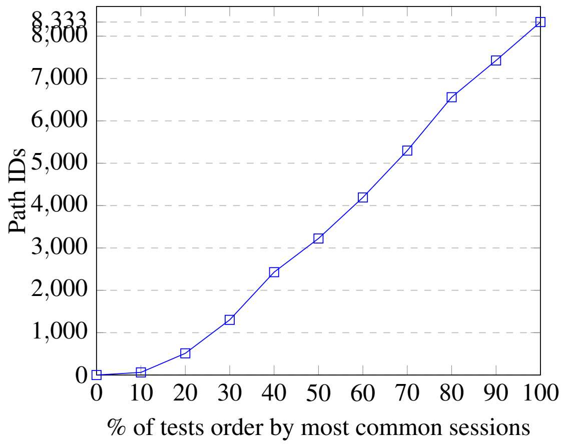


Figure 4.3: Relationship between the percentage of the total tests ordered by most common sessions and the number of total covered Path IDs

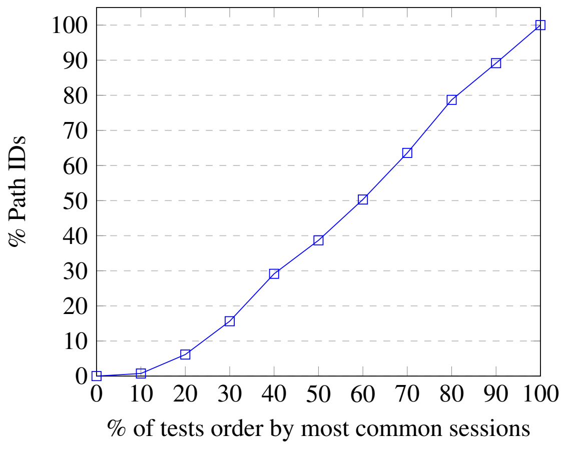


Figure 4.4: Relationship between the percentage of the total tests ordered by most common sessions and the percentage of total covered Path IDs

4.4.1.2 Order by Diverse Path ID

This approach aims to select test cases based on the Path IDs that they test, using the *Diverse* filter, available on the Analysis App. Ordering the test cases by the value they bring to the test suite allows reaching better results with fewer test cases, removing redundant sessions that are not testing different elements. This method was previously explained in section 3.2.2.4 of the Proposed Approach chapter 3.

As represented in Figure 4.5 and in Figure 4.6, the total coverage of the collected Path IDs is achieved with 53.63% of the 4900 sessions, what leads to 2628 test cases. As we can see in Figure 4.7, with 10% of the test cases it is already possible to achieve 42.44% of coverage of the considered Path IDs. With half of the test suite, 83% of the Path IDs are covered.

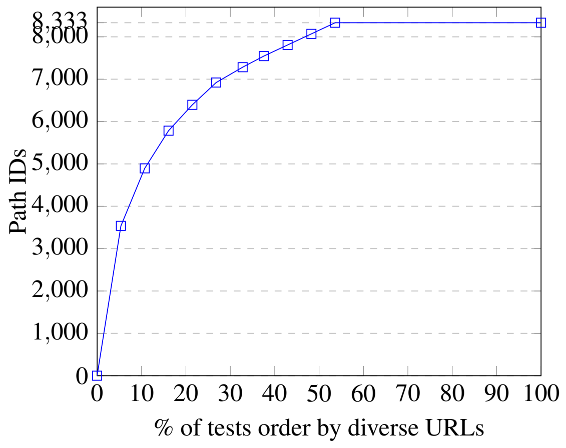


Figure 4.5: Relationship between the percentage of the total tests ordered by *Diverse* Path ID and the number of total covered Path IDs

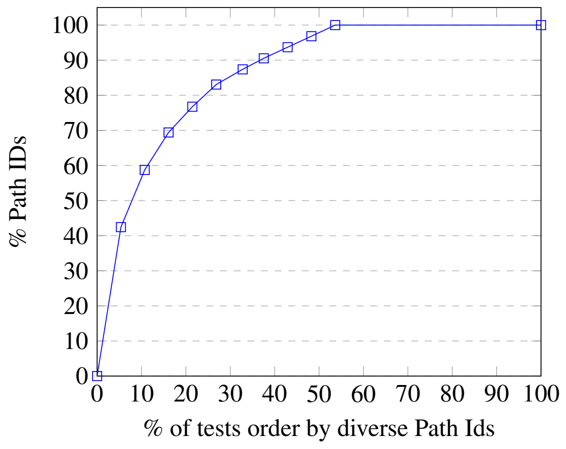


Figure 4.6: Relationship between the percentage of the total tests ordered by *Diverse* Path ID and the percentage of covered Path IDs

Suite	Total	Covered Path IDs									
		10%*	20%*	30%*	40%*	50%*	60%*	70%*	80%*	90%*	100%*
1	2913	941	1863	2700	3514	4363	5248	5930	6736	7623	8333
2	2908	1063	1934	2750	3675	4506	5284	6101	6814	7581	8333
3	2923	956	1878	2639	3490	4392	5245	6028	6855	7593	8333
4	2922	799	1730	2598	3369	4257	5079	5952	6759	7509	8333
5	2909	886	1765	2550	3271	4187	5061	5885	6756	7527	8333
6	2910	949	1836	2844	3656	4472	5233	6017	6773	7578	8333
7	2903	1008	1891	2690	3584	4343	5171	5950	6733	7570	8333
8	2932	975	1929	2802	3522	4281	5126	5979	6800	7578	8333
9	293	1932	1889	2715	3662	4493	5358	6126	6861	7668	8333
10	2905	935	1816	2791	3539	4356	5227	5997	6826	7587	8333
$\bar{x}$	2916	944	1853	2708	3528	4365	5203	5997	6791	7581	8333
σ	10,60	69,92	66,94	93,24	130,04	105,10	93,29	74,83	46,89	44,41	0

*Percentage of the total test cases of each test suite.

Table 4.3: Path IDs tested on each suite generated by the *Random approach*.

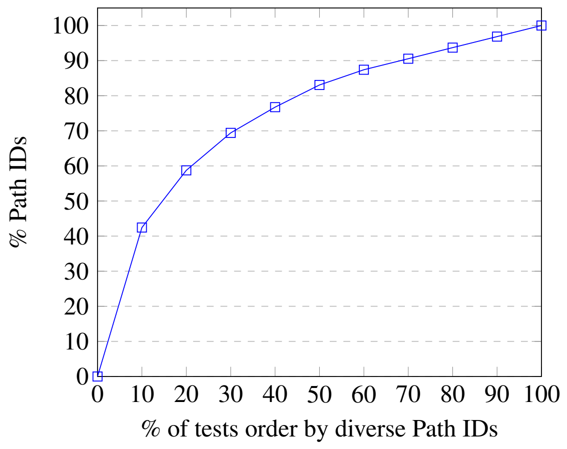


Figure 4.7: Relationship between the percentage of the selected tests ordered by *Diverse* Path ID and the percentage of covered Path IDs

4.4.1.3 Random Approach

The *Random approach* aims to pick randomly a session, get the Path IDs covered by the session, and add it to the test suite. To have consistent results, 10 test suites were generated and analysed, which results are presented in Table 4.3.

Analyzing the coverage with the average of the data retrieved in the 10 test suites, we conclude that 100% of coverage is achieved when 59.48% of test cases are performed. The Figure 4.8 and Figure 4.9 represent the test cases selected randomly and their coverage. It actually improves linearly, as it is proved in Figure 4.10.

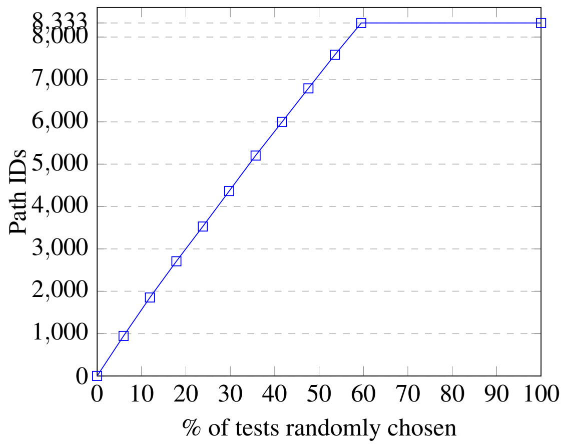


Figure 4.8: Relationship between the percentage of tests randomly chosen and the number of covered Path IDs

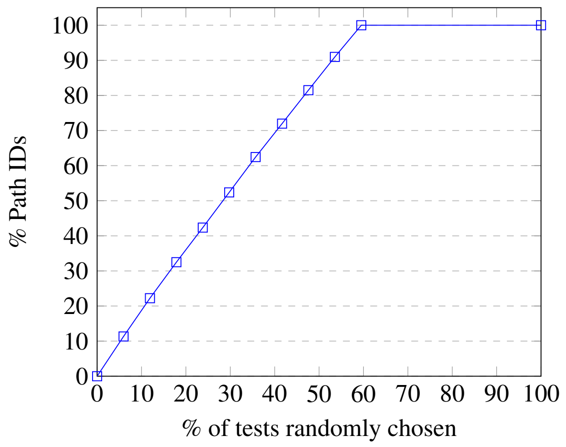


Figure 4.9: Relationship between the percentage of tests randomly chosen and the percentage of covered Path IDs

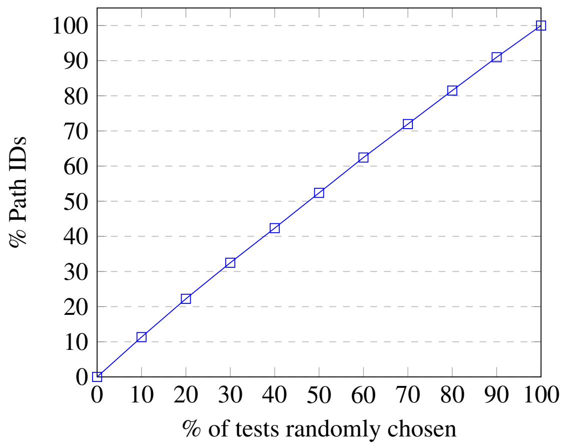


Figure 4.10: Relationship between the percentage of selected tests randomly chosen and the percentage of covered Path IDs

4.4.2 URLs

To get all URLs of the application under test, the *wget* tool[2], a free software package for retrieving files from Web pages was used and 17865 pages were retrieved. Therefore, this data was analysed next: all the GET parameters sent on URLs were removed as well as the pages that lead to files (such as images or pdf files). Finally, 4612 different URLs were obtained under the `ipvc.pt` domain. The total saved interactions would span 1111 different URLs, according to the collected data. That leads to a 24.01% coverage.

However, from the 4612 pages, 1067 URLs belong to the calendar, since each day view is represented by one different URL, as well as each week, month or year present on the calendar. Although URLs are different, the page structure is the same between each type. So that, if we only considered four different pages, one for each time unit, instead of 1067, the coverage will raise to 31.30%. From the collected data, there is no interaction in any of the calendar pages, which denotes that the calendar area is not much visited by the users.

The *node* section represents a similar problem. There are pages with the same structure which URL keeps the following structure: `http://www.ipvc.pt/node/2579`, changing only the ID on the end of the URL. From the 4612 URLs explored, 1410 seem to represent the equal structured page, varying only on the left-sided menu, having two different versions. If we only considered two pages, one for each menu version, the coverage increases to 51.89%. There are 125 interactions on this type of pages on our collected data, representing 11 different news URLs. Furthermore, it was found that there are more equal structured pages, however, without the same URLs structure that was previously referred. This makes us believe that there are more pages in the 2150 left pages with the same structure that, if removed, would lead to a coverage improvement. Unfortunately, since we do not have this kind of information, it is not possible to make this deeper analysis.

Next, it will be presented three different approaches that were followed in order to select test cases according to different criteria - most common sessions, sessions that test more diverse URLs, *Random approach* - and analyse the respective coverage of the collect URLs (1111). This process is similar to the one followed to analyse the Path IDs coverage.

Moreover, it was selected a subset of the most visited URLs, choosing the ones which have more than 100 interactions. As a result, we have a set of 24 URLs, presented in the Table 4.1, that will be used to analyse the coverage of our tests in this smaller sample, representing the most visited pages.

4.4.2.1 Order by Most Common Sessions

Firstly, test cases were ordered by the most common session to the less common one. Only when testing the 4900 possible sessions, the 1111 collected URLs are covered. As it can be noticed in the graphs, Figure 4.11 and Figure 4.12, the result is close to linear, which means that, for instance, 50% of the tests leads close to 50% URLs coverage.

[2]https://www.gnu.org/software/wget/

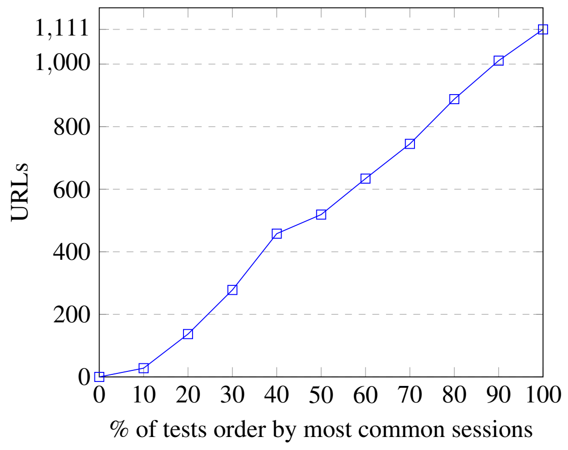


Figure 4.11: Relationship between the percentage of total tests order by most common session and the number of covered URLs

In addition, since the most common sessions are usually very small (easier to be most performed), not allowing to go through many different pages, we filtered sessions by length, which combined with the most common order result in different conclusions. For instance, selecting from the most common sessions, the ones whose length is bigger than 5 interactions achieve a 74.7% coverage (830 in 1111 URLs) with 1089 tests cases. If we increase the length limit to 10, with 381 test cases, it is possible to achieve 58.7% coverage (652 in 1111 URLs).

Moreover, focusing on the most common subset of URLs, represented in Table 4.1, we can achieve the total coverage, 100% of the 24 pages, with 165 sessions bigger than 5 interactions. The total coverage is reached with 41 test cases bigger than 10 nodes. Without length filter, the 24 URLs will be covered after 984 tests (20.08% of total tests). While the length filter does not allow to achieve total coverage on the total set of URLs, in this subset, the results are improved when applied the filter.

4.4.2.2 Order by Diverse URLs

Secondly, if we use the *Diverse* URLs function of the analysis tool to order the test cases, we get a differently ordered test suite. This approach aims to select test cases depending on the value they add to the test suite, i.e a test case is added to the suite when is the one that adds more different URLs to the set of URLs covered by the already selected test cases. Redundant sessions are removed from the test suite, allowing to test the same pages with fewer test cases.

As we can see in the Figure 4.13 and in the Figure 4.14, with only 10.4% of the total test cases (4900), which means 510 test cases, we achieve total coverage of the 1111 collected URLs. Focusing on the subset of most visited URLs, it is possible to achieve the coverage of the 24 pages with 73 test cases (1.48% of the total sessions). In Figure 4.15, we can see the URLs distribution by the 510 test cases that lead to total coverage of the collected pages. Actually, almost half of the URLs are tested with less than 20% of the 510 tests.

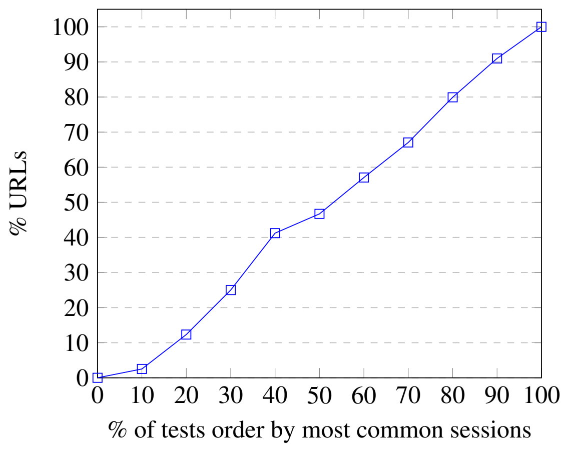


Figure 4.12: Relationship between the percentage of total tests order by most common session and the percentage of covered URLs

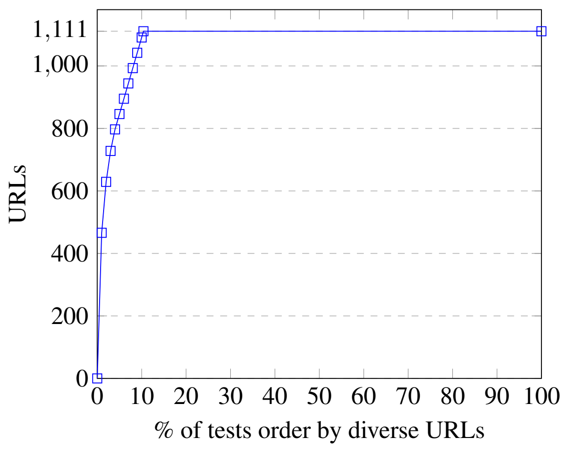


Figure 4.13: Relationship between the percentage of total tests ordered by *Diverse* URLs and the number of total covered URLs

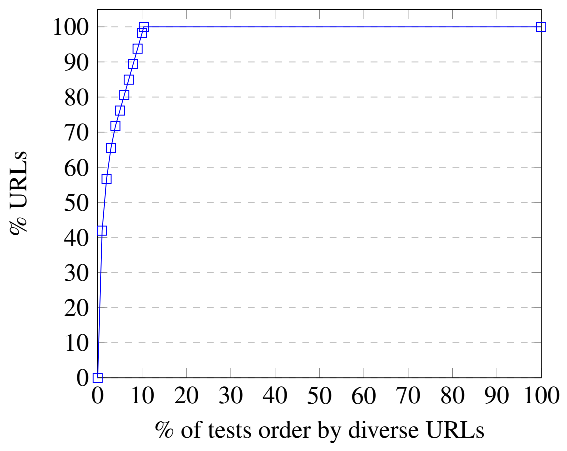


Figure 4.14: Relationship between the percentage of total tests ordered by *Diverse* URLs and the percentage of covered URLs

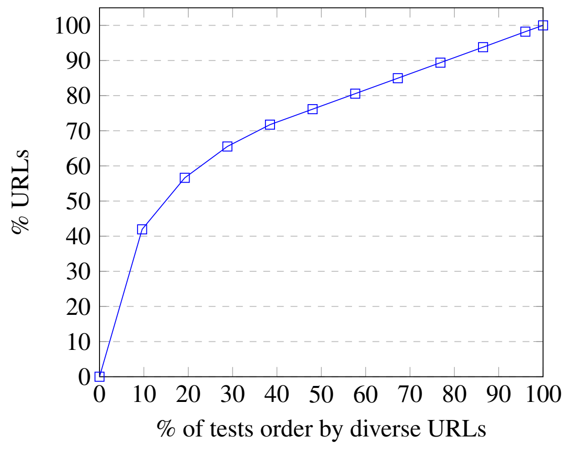


Figure 4.15: Relationship between the percentage of the selected tests ordered by *Diverse* URLs and the percentage of covered URLs

4.4.2.3 Random Approach

Last but not least, the *Random approach* selects, randomly, a session to add to the test suite. This technique revealed interesting results, since it reaches the total coverage of the collected pages with 14.8% test cases, which means 725 test cases of the 4900 possible sessions, as it is presented in Figure 4.16 and in Figure 4.17. Focusing on the selected test cases, represented in Figure 4.18, we can notice a graph very close to a linear graph.

Analyzing the coverage of the subset of the most visited (Table 4.1), it can be concluded that, with 123 test cases, the 24 URLs would be covered, representing only 2.51% of the total sessions.

This *Random approach* can origin different test suites, with slight differences, every time a new one is created. In order to get representative data, 10 test suites were created and retrieved the coverage of each one, from 10% to 10%, allowing us to compare the values across the 10 suites in 10 points through them. These values are represented in the Table 4.4. The final values were calculated as the arithmetic mean of the values for the 10 suites. Only these values were used to build the graphs and to make conclusions about this technique.

In general, the number of covered URLs in each percentage of test cases is very close between test suites. That can be proved by calculating the standard deviation, which shows a small range of values. Standard deviation is higher in the subset of the most visited URLs because we are looking for specific things. Since the test cases are randomly selected, these specific URLs can actually appear at any order in the test suite.

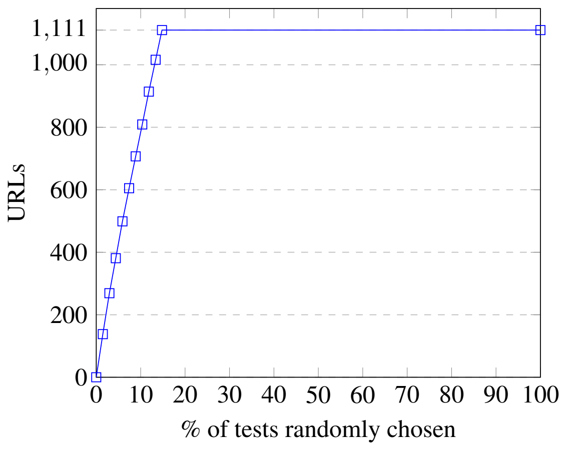


Figure 4.16: Relationship between the percentage of tests randomly chosen and the number of covered URLs

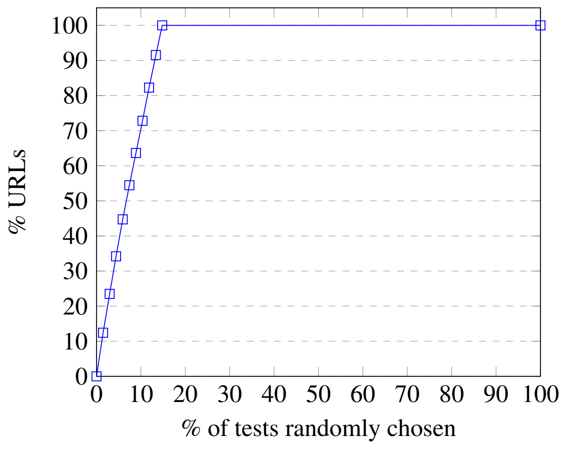


Figure 4.17: Relationship between the percentage of total tests randomly chosen and the percentage of covered URLs

Suite	Tests		Covered URLs									
	Total	Set**	10%*	20%*	30%*	40%*	50%*	60%*	70%*	80%*	90%*	100%*
1	725	104	145	267	377	498	610	699	802	911	1019	1111
2	728	189	137	266	389	504	604	697	797	896	1008	1111
3	751	166	141	244	341	451	556	665	785	907	1008	1111
4	737	135	128	261	376	488	599	704	811	924	1018	1111
5	706	80	146	256	381	513	624	728	819	916	1016	1111
6	725	89	145	276	406	515	614	715	816	911	1018	1111
7	726	183	139	243	397	511	611	716	822	924	1022	1111
8	721	149	132	253	382	493	607	712	817	926	1027	1111
9	715	68	144	259	364	501	617	717	809	919	1013	1111
10	722	72	131	277	403	520	615	721	820	909	1018	1111
$\bar{x}$	725	123.5	138	269	381	499	605	707	809	914	1016	1111
σ	12.10	42.96	6.90	10.48	18.55	19.44	18.73	17.60	11.40	9.60	6.55	0

*Percentage of the total test cases of each test suite.
** The most visited pages subset, represented in Table 4.1.
Table 4.4: Elements tested on each URL of the 1% most visited set

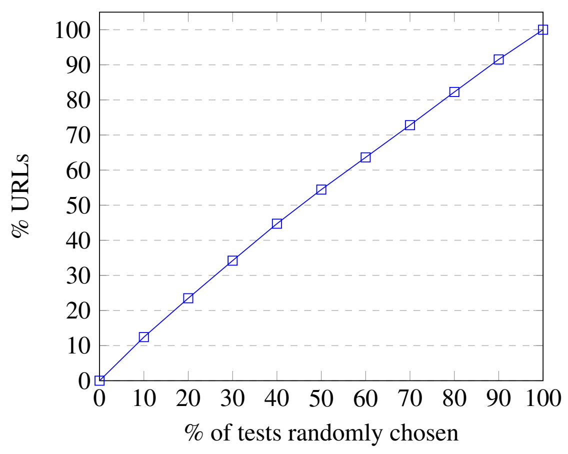


Figure 4.18: Relationship between the percentage of the selected tests randomly chosen and the percentage of covered URLs

4.5 Summary

The coverage in `www.ipvc.pt` is not very high because this Web site has a lot of historic information, collecting a lot of not visited and old pages. The way the Web site is organized also ends in a lot of different URLs. In fact, the structure and the amount of data can impact our results. Since it was difficult to collect usage data in all pages as well as to interact with all elements, the test coverage is affected by that.

Focusing in the 11 most visited URLs we can notice by observing the data represented in Table 4.2 that the interactive elements coverage is not very high since the average for that URLs subset is

Approach	Path IDs	URLs	
		All	Subset
Common	100%	100%	20.08% (0.84%)
Diverse	53.63%	10.4%	1.48%
Random	59.48%	14.8%	2.51%

Table 4.5: Percentage of tests to achieve total coverage following different approaches

36.94%. Pages are mostly informative and the interactive elements that they have are mainly links to other pages, so the users likely interact with only a few of them, explaining why the percentage of coverage is not very high. After pre-processing URLs, the maximum pages coverage achieved was 24.01%, which means that we only have collected data in 24.01% of the pages.

After that, we considered only captured data to get the coverage of the generated test cases against the retrieved data during the capture process. Considering all Path IDs and URLs collected, from the three different approaches presented (*Common*, *Diverse* and *Random*), the *Diverse approach* was the one which achieves total coverage with fewer test cases, either on elements and pages. Since this solution analyses which test case brings more value to add to the test suite before adding any, it would be expected that it would achieve better results. With 53.63% of the total sessions, it would be possible to achieve total coverage of the Path IDs. URLs would all be covered with only 10.4% of test cases.

The proposal that aims to order test cases by common sessions always needs to run the total tests in order to get 100% of coverage. This can be explained by the existence of some interactions, not much performed, that by being uncommon stand at the end of the test suite, preventing from achieving total coverage before running the last test cases. That can explain why *Random approach* performs better, because the less common sessions have a higher chance of being performed before, achieving first the total coverage. Actually, the *Random approach* achieves results much close to the *Diverse approach*, with less effort to build the test suite. These values are centralized in the Table 4.5.

However, in the most visited subset, the total coverage is achieved with fewer test cases when ordered by the most common session and combined to a length filter. Since it is only considered a set of items, there is no need to perform the whole test suite to achieve particularly uncommon interactions. This proves that each approach has its own target, depending on the testing goal and the acceptable effort, i.e the number of test cases intended to run.

When we only focus on the collected data, we could see that it is possible to achieve high levels of coverage, even total, without great effort, using the most suitable approach to select test cases. So, the coverage actually depends on the collected usage: the richer collected data is, the easier is to achieve high coverage of the application.

Chapter 5

Conclusions and Future Work

In this work, we were motivated by finding a way of generating automated GUI tests with less effort, trying to fight one of the main reasons why companies still rely on manual testing: to avoid taking time on learning how to use tools, building models or scripts and maintaining them. However, adopting automated tests allow running them every time the application changes, having a stronger quality control than when performing manual testing. We presented MARTT - Mining Automated Regression Testing Tool: a solution which generates automated GUI tests for Web applications based on real usage, properly adapted to the application needs and reflecting real users behaviour.

This document has presented the state of the art regarding automated GUI test cases generation techniques and regression testing, focusing especially on usage information and on generating test cases approaches that use it to derive test cases. Trying to provide some context to this work, the state of the art started by presenting four methods to generate automated GUI tests: random testing, capture and replay, scripting and model-based testing. We proposed to include our approach in model-based testing techniques, considering our captured usage data set as a model since we analyse that data and select the most important subsets with the purpose of generating test cases from that. After that, usage information was outlined, presented the two main methods to collect information: in the server side or in the client side, by Web server logs or by page tagging method, accordingly. To conclude the state of the art, it is provided context about regression testing process and its three main phases: test case generation, test case prioritization and test case execution. It is important to notice that we only focused on test case generation techniques based on usage information since they are the ones closest to our approach.

Focusing on how MARTT was implemented, three main phases were described. The first one relies on capturing data needed to generate test cases, depending on the client-side script (responsible for collecting data) and the API (responsible to communicate with the database to save the collected interactions). Then, in a second phase, the data is analyzed according to defined criteria with the purpose of selecting efficient and useful test cases from the whole test sessions collected. Lastly, in the third phase, test scripts are generated from the files retrieved in the previous step.

The databased used in this work was *Neo4j*: a graph database with native graph storage and processing. It was useful since the saved data have strong relationships and querying them was more intuitive. The visual browser that *Neo4j* provides is really interesting because the nodes appear connected by session, giving knowledge about sessions just at a first glance. At the beginning of this work, Neo4j seemed a better option when compared with a relational database. Indeed, the way data is modelled would not fit properly in a relational database and the model would have to be adapted. The only limitations of Neo4j were in *Neo4j* algorithms library that did not allow us to analyse the data in a way we expected to do: extracting the most common sessions easily. To solve that problem, we implemented the Levenshtein algorithm to get the most common sessions.

In order to validate our work, we applied it on a real Web application [1]. The results were presented in chapter 4. We analysed the global pages coverage and have achieved a maximum value of 24.01%, after having pre-processed URLs, joining the ones with the same structure. Selecting the 11 most visited URLs and calculating the coverage of their elements, we can conclude that, because pages are mostly informative and static, the elements coverage is not very high, resulting in an average cover of 36.94%.

The deeper coverage study presented was regarding Path IDs and URLs collected. It was used three different approaches to calculate test cases coverage against the captured data: select test cases according to the most common sessions, select test cases according to the most diverse items they test or select randomly. We noticed that the better approach to achieve total coverage, either on Path IDs or URLs, was the *Diverse approach* since it chooses the test case according to the value it adds to the test suite, comparing it to the ones that were already selected. The random approach revealed good results since test cases are selected without none criteria, staying behind *Diverse approach* by a slight difference. The approach which selects test cases based on the most common sessions only achieve total coverage when running all test cases. However, it suits better to the scenario which considers a subset of most common URLs. We could conclude that the coverage actually depends on the collected usage. If the captured data represent the whole website, it is easy to achieve high coverage when executing the selected test cases.

After the investigation concluded we are able to answer the research questions proposed on problem section (1.3).

- **RQ1. Are we able to generate an executable test suite from real usage data?**

  yes, we are. The case study validated our work, meeting the goal of having test cases generated after retrieving real data from the `http://www.ipvc.pt/`.

- **RQ2. How much coverage of the application, regarding pages, can we get with all sessions?**

  The coverage achieved was 51.89% considered the same structured pages as one and 24.01% without processing the data. This means that the captured interactions were performed in

[1] http://www.ipvc.pt/

24.01% of the pages. The structure and nature of the application that was under test did not help to reach more pages. This approach would get better results in applications that are widely covered by the users when interacting with it. More different collected data means more application coverage.

- **RQ3. How efficient is the reduction of test cases based on the collected sessions, regarding elements and pages coverage?**

  The reduction of test cases was especially efficient using the *Diverse approach* since, with 53.63% of the whole test cases we can achieve the same Path ID coverage as when executing all sessions. Regarding URLs, with only 10.4% of total test cases, we can cover the same pages as it would be achieved with all test sessions.

Our study may not be completely free of errors. Regarding internal validity, we have used a Web site, "IPVC", that has some particularities. For instance, dynamic behaviour associated to the XPath, which means that the same element may be identified differently in different sessions. This has an impact on the elements coverage analysis. In order to mitigate this problem, we have performed a pre-processing that had removed dynamic IDs from the XPath of elements affected by this dynamic behaviour. Also, when parameters are passed through the URL, some Web pages may be detected as distinguish pages. This may have an impact on the URL coverage analysis. To mitigate this problem we have removed the parameters from the URLs.

Regarding external validity, our results may not generalize to other Web sites. In particular, if the Web site under analysis has a different nature, for instance, with less historical data related to older news, the results achieved may be different. In this particular case, we would expect to exercise a higher percentage of interactive UI elements. We could mitigate this problem by filtering URLs with old content and remove them from the URL coverage analysis. We did not perform that mitigation action because the Web site used is public, we are not the owners of the Web site, and we do not have this type of information, neither a way to get it.

In this work we used XPath, but we had some problems with dynamic IDs generated each time the application was reloaded, led to equal elements with different XPaths. The data were processed to reducing the impact of this issue. As future work, we would like to improve the way GUI elements are identified to deal better with dynamic behaviour. The way inputs are being captured may also be improved. If the input type was saved, generating test inputs when running the test script would be an easier and not so vague process since we would have information like *number*, *month*, *email* that would help. However, it is important to have in mind that no personal information must be disclosed.

The application that allows filtering the test cases could also be improved, not only the UI but also more features could be added: file download could be made all at the same time when more than one test cases are selected, instead of having individual downloads; sessions could have a preview mode before being downloaded, to help tester to select the more suitable ones; more algorithms to find most common sessions could be added.

References

[1] A. Abdurazik and J. Outt. Using UML Collaboration Diagrams for Static Checking and Test Generation. *Proceedings of the 3rd International Conference on the Unified Modelling Language*, page 383–395, 2000.

[2] E. Alégroth. Random Visual GUI Testing: Proof of Concept. In *SEKE*, 2013.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Federico. Rich Internet Application Testing Using Execution Trace Data. *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 274–283, 2010.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, One edition, 2008.

[5] M. Barnett, K. Leino, M. Rustan, and W. Schulte. The Spec# Programming System: An Overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[6] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings 11th International Conference on World Wide Web WWW02*, page 654–668, 2002.

[7] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. *Electronic Notes in Theoretical Computer Science 116*, page 85–97, 2004.

[8] International Software Testing Qualifications Board. Regression testing. Available at http://glossary.istqb.org/search/regression%20testing, November 2018.

[9] I. Burnstein. *Practical Software Testing*. Springer, first edition, 2003.

[10] Y. Chen, R. Probert, and D. P. Sims. Specification-based Regression Test Selection with Risk Analysis. In *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '02, pages 1–14. IBM Press, 2002.

[11] V. Dallmeier, B. Pohl, M. Burger, M. Mirold, and A. Zeller. WebMate: Web Application Test Generation in the Real World. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 413–418, 2014.

[12] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by Cluster Analysis of Execution Profiles. *Proceedings - International Conference on Software Engineering*, pages 339–348, 2001.

[13] S. Elbaum, S. Karre, and G. Rothermel. Improving Web Application Testing With User Session Data. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 49–59, May 2003.

[14] A. M. Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging Existing Tests in Automated Test Generation for Web Applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 67–78, 2014.

[15] J. Fernandes and A. D. Fonzo. When to Automate Your Testing (and When Not To). Available at https://www.oracle.com/technetwork/cn/articles/when-to-automate-testing-1-130330.pdf, December 2018.

[16] L. Fernandez-Sanz and S. Misra. Practical Application of UML Activity Diagrams for the Generation of Test Cases. *Proceedings of the Romanian Academy - Series A: Mathematics, Physics, Technical Sciences, Information Science*, 13:251–260, 07 2012.

[17] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Third edition, 2014.

[18] S. Gnesi, D. Latella, and M. Massink. Formal Test-case Generation for UML Statecharts. *Proceedings of the 9th IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e- Engineering. Vol. 42*, page 75–84, 2004.

[19] N. Gupta, V. Yadav, and M. Singh. Automated Regression Test Case Generation for Web Application. *ACM Computing Surveys*, 51(4):1–25, 2018.

[20] P. Haar and D. Michaëlssion. *Automated GUI Testing: A Comparison Study With A Maintenance Focus*. PhD thesis, Chalmers University of Technology and University of Gothenburg, SE-412 96 Gothenburg Sweden, 2018.

[21] J. Hao and E. Mendes. Usage-based Statistical Testing of Web Applications. In *Proceedings of the 6th International Conference on Web Engineering*, ICWE '06, pages 17–24, New York, NY, USA, 2006. ACM.

[22] M. J. Harrold, R. Gupta, and M. L. Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[23] S. Jain, R. Rawat, and B. Bhandari. A survey Paper on Techniques and Applications of Web Usage Mining. In *2017 International Conference on Emerging Trends in Computing and Communication Technologies (ICETCCT)*, pages 1–6, 2017.

[24] U. P. Jyothi1, S. Bonthu, and B. V. Prasanthi. A Study on Raise of Web Analytics and its Benefits. In *International Journal of Computer Sciences and Engineering*, pages 59–64, 2007.

[25] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, Nov 2001.

[26] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996.

[27] D. Leon, A. Podgurski, and L. J. White. Multivariate Visualization in Observation-based Testing. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pages 116–125, June 2000.

[28] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[29] Y. F. Li, P. K. Das, and D. L. Dowe. Two Decades of Web Application Testing - A Survey of Recent Advances. *Information Systems*, 43:20–54, 2014.

[30] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 111–122, May 2015.

[31] S. Mallaiah and S. Manjula. A Systematic Strategy for Extracting Frequent Items Through Associate Analysis. In *Proceedings of International Conference on "Information Science & Technology for Sustainability & Innovation"*, 2015.

[32] A. M. Memon and M. E. Pollack. Plan Generation for GUI Testing. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 226–235, 2000.

[33] R. M. L. M. Moreira and A. C. R. Paiva. PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-Based GUI Testing. In *SE '14 Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 1–5, 2014.

[34] R .M. L. M. Moreira, A. C. R. Paiva, and A. Memon. A Pattern-based Approach for GUI Modeling and Testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 288–297, Nov 2013.

[35] R .M. L. M. Moreira, A. C. R. Paiva, M. Nabuco, and A. Memon. Pattern-based GUI testing: Bridging the Gap Between Design and Quality Assurance. *Software Testing Verification and Reliability*, 2017.

[36] I. C. Morgado and A. C. R. Paiva. The iMPAcT Tool: Testing UI Patterns on Mobile Applications. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 876–881, Nov 2015.

[37] I. C. Morgado, A. C. R. Paiva, and J. Faria. Reverse Engineering of Graphical User Interfaces. In *Proceedings of ICSEA 2011 : The Sixth International Conference on Software Engineering Advances*, pages 293–298, 2011.

[38] I. C. Morgado and A. C.R. Paiva. Impact of Execution Modes on Finding Android Failures. *Procedia Computer Science*, 83:284 – 291, 2016. The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops.

[39] K. Naik and P. Tripathy. *Software Testing and Quality Assurance, Theory and Practice*. Wiley-Spektrum, One edition, 2008.

[40] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. Characterization of Model-based Software Testing Approaches. (December 2006):114, 2007.

[41] S. P. Nina, M. Rahman, K. I. Bhuiyan, and K. E. U. Ahmed. Pattern Discovery of Web Usage Mining. In *2009 International Conference on Computer Technology and Development*, pages 499–503, 2009.

[42] A. C. R. Paiva, J. C. P. Faria, and R. F. A. M. Vidal. Towards the Integration of Visual and Formal Models for GUI Testing. *Electronic Notes in Theoretical Computer Science*, pages 99 – 111, 2007. Proceedings of the Third Workshop on Model Based Testing (MBT 2007).

[43] A. C. R. Paiva, J. Garcia, A. Restivo, and P. Silva. Automatic Test Case Generation from Usage Information. In *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, Sep 2018.

[44] Regression testing. Available at http://softwaretestingfundamentals.com/regression-testing/, May 2019.

[45] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 25–34, May 2001.

[46] M. Riebisch, I. Philippow, and M. Gotze. UML-based Statistical Test Case Generation. *Objects, Components, Architecture, Services for Applications for a Networked World*, page 394–4111, 2002.

[47] I. Sommerville. *Software Engineering*. Pearson, Ninth edition, 2011.

[48] H. Srikanth, L. Williams, and J. Osborne. System Test Case Prioritization of New and Regression Test Cases. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–, Nov 2005.

[49] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A Capture/Replay Tool for Observation-based Testing. *SIGSOFT Softw. Eng. Notes*, 25(5):158–167, August 2000.

[50] The Database Model Showdown: An RDBMS vs. Graph Comparison. Available at https://neo4j.com/blog/database-model-comparison/, May 2019.

[51] A. Torsel. Automated Test Case Generation for Web Applications from a Domain Specific Model. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 137–142, 2011.

[52] C. Trammell. Quantifying the Reliability of Software: Statistical Testing Based on a Usage Model. In *Proceedings of Software Engineering Standards Symposium*, pages 208–218, Aug 1995.

[53] F. Tsui, O. Karam, and B. Bernal. *Essentials of Software Engineering*. Jones & Bartlett Learning, Third edition, 2014.

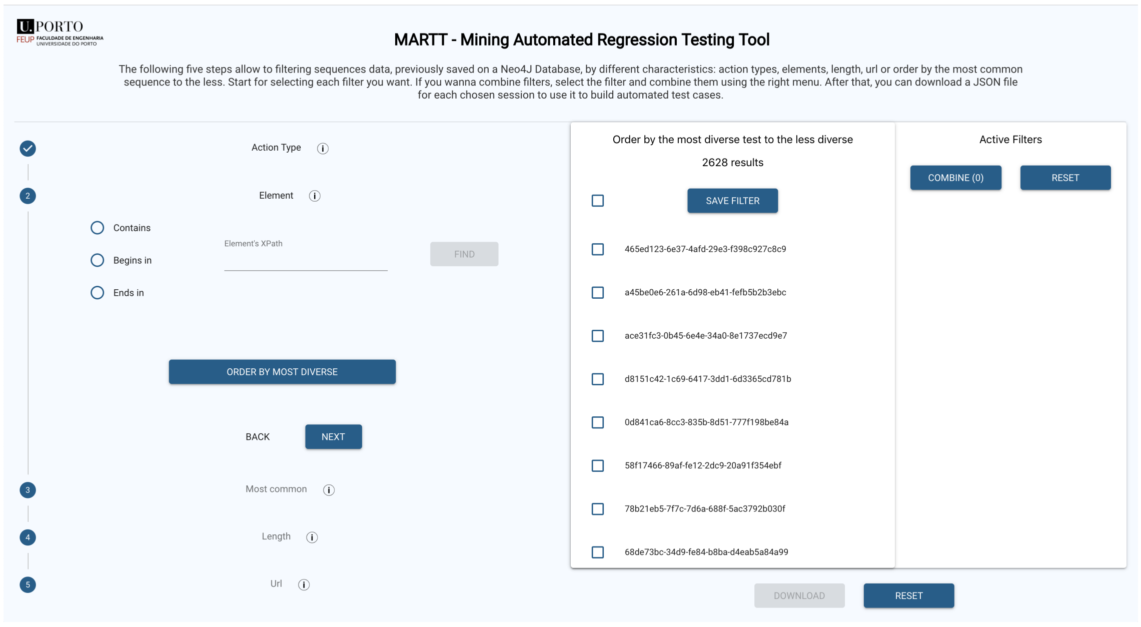
[54] T. Wetzlmaier and R. Ramler. Hybrid Monkey Testing: Enhancing Automated GUI Tests with Random Test Generation. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, 2017.

[55] Q. Zhongsheng. Test Case Generation and Optimization for User Session-based Web Application Testing. *JCP*, 5:1655–1662, 2010.

Appendix A

MARTT - UI

Some screenshots of our UI application are here presented. These screenshots were captured with real data of our case study.


MARTT - Mining Automated Regression Testing Tool

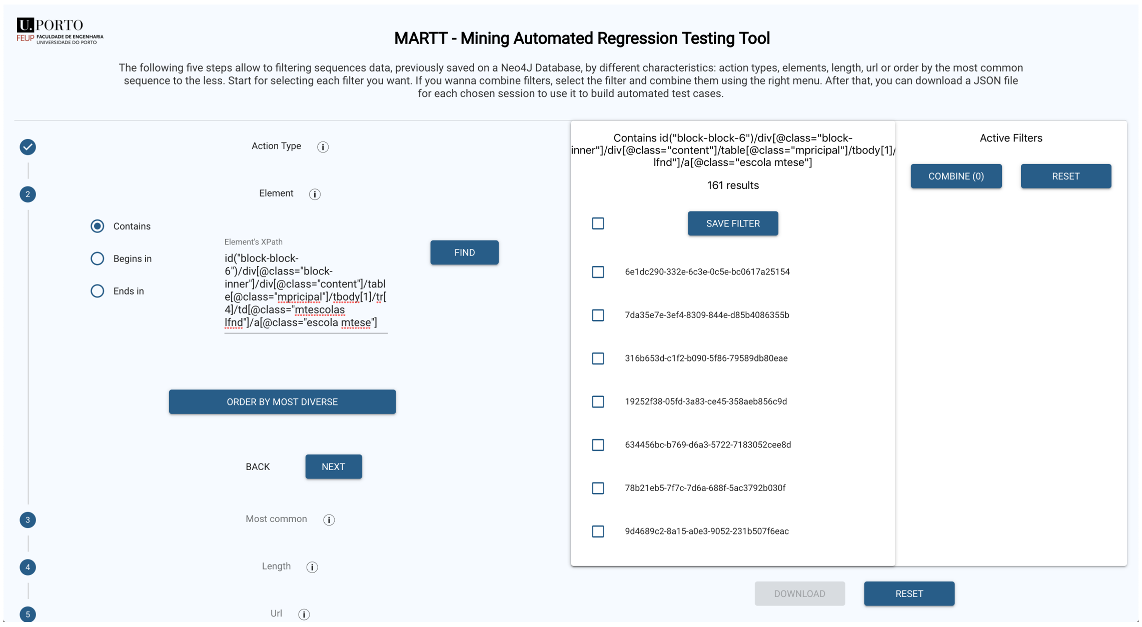
The following five steps allow to filtering sequences data, previously saved on a Neo4J Database, by different characteristics: action types, elements, length, url or order by the most common sequence to the less. Start for selecting each filter you want. If you wanna combine filters, select the filter and combine them using the right menu. After that, you can download a JSON file for each chosen session to use it to build automated test cases.

Order by the most diverse test to the less diverse

2628 results

465ed123-6e37-4afd-29e3-f398c927c8c9
a45be0e6-261a-6d98-eb41-fefb5b2b3ebc
ace31fc3-0b45-6e4e-34a0-8e1737ecd9e7
d8151c42-1c69-6417-3dd1-6d3365cd781b
0d841ca6-8cc3-835b-8d51-777f198be84a
58f17466-89af-fe12-2dc9-20a91f354ebf
78b21eb5-7f7c-7d6a-688f-5ac3792b030f
68de73bc-34d9-fe84-b8ba-d4eab5a84a99

Figure A.1: UI application with *Diverse* filter active on elements.



MARTT - Mining Automated Regression Testing Tool

The following five steps allow to filtering sequences data, previously saved on a Neo4J Database, by different characteristics: action types, elements, length, url or order by the most common sequence to the less. Start for selecting each filter you want. If you wanna combine filters, select the filter and combine them using the right menu. After that, you can download a JSON file for each chosen session to use it to build automated test cases.

Action Type
2 Element
[✓ Contains] Begins in Ends in
Element's XPath: id("block-block-6")/div[@class="block-inner"]/div[@class="content"]/table[@class="mpricipal"]/tbody[1]/tr[4]/td[@class="mtescolas lfnd"]/a[@class="escola mtese"]
3 Most common
4 Length
5 Url

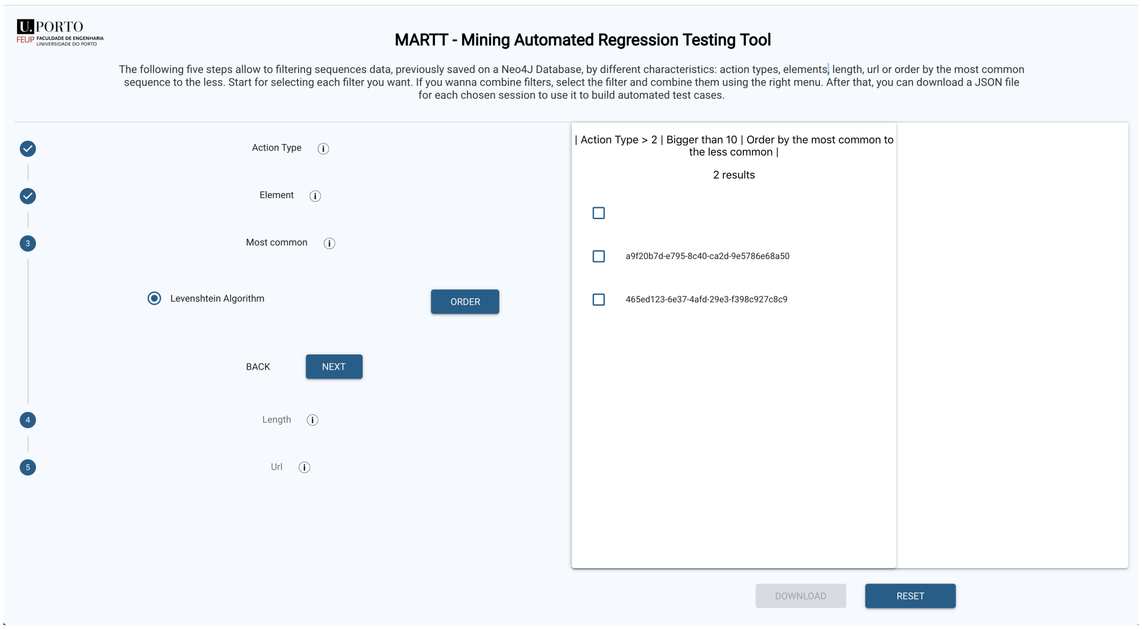
Contains id("block-block-6")/div[@class="block-inner"]/div[@class="content"]/table[@class="mpricipal"]/tbody[1]/lfnd"]/a[@class="escola mtese"]

161 results

- 6e1dc290-332e-6c3e-0c5e-bc0617a25154
- 7da35e7e-3ef4-8309-844e-d85b4086355b
- 316b653d-c1f2-b090-5f86-79589db80eae
- 19252f38-05fd-3a83-ce45-358aeb856c9d
- 634456bc-b769-d6a3-5722-7183052cee8d
- 78b21eb5-7f7c-7d6a-688f-5ac3792b030f
- 9d4689c2-8a15-a0e3-9052-231b507f6eac

Active Filters

Figure A.2: UI application with *Element* filter active.



MARTT - Mining Automated Regression Testing Tool

The following five steps allow to filtering sequences data, previously saved on a Neo4J Database, by different characteristics: action types, elements, length, url or order by the most common sequence to the less. Start for selecting each filter you want. If you wanna combine filters, select the filter and combine them using the right menu. After that, you can download a JSON file for each chosen session to use it to build automated test cases.

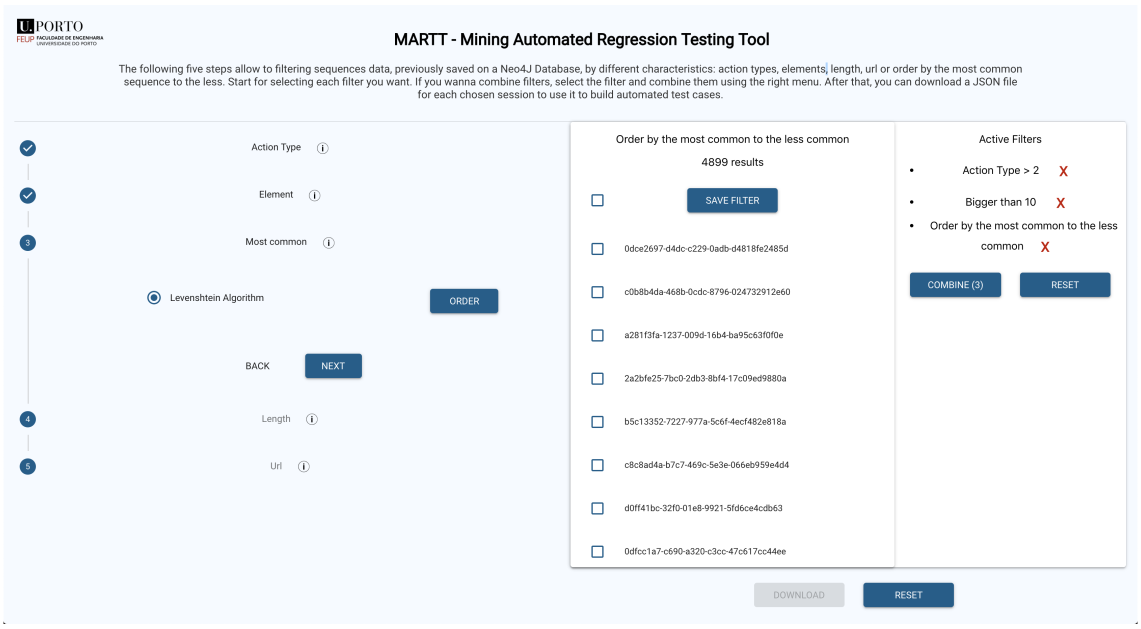
Action Type
Element
3 Most common
[✓ Levenshtein Algorithm]
4 Length
5 Url

| Action Type > 2 | Bigger than 10 | Order by the most common to the less common |

2 results

- a9f20b7d-e795-8c40-ca2d-9e5786e68a50
- 465ed123-6e37-4afd-29e3-f398c927c8c9

Figure A.3: UI application with *Action Type* filter active


MARTT - Mining Automated Regression Testing Tool

The following five steps allow to filtering sequences data, previously saved on a Neo4J Database, by different characteristics: action types, elements, length, url or order by the most common sequence to the less. Start for selecting each filter you want. If you wanna combine filters, select the filter and combine them using the right menu. After that, you can download a JSON file for each chosen session to use it to build automated test cases.

Order by the most common to the less common

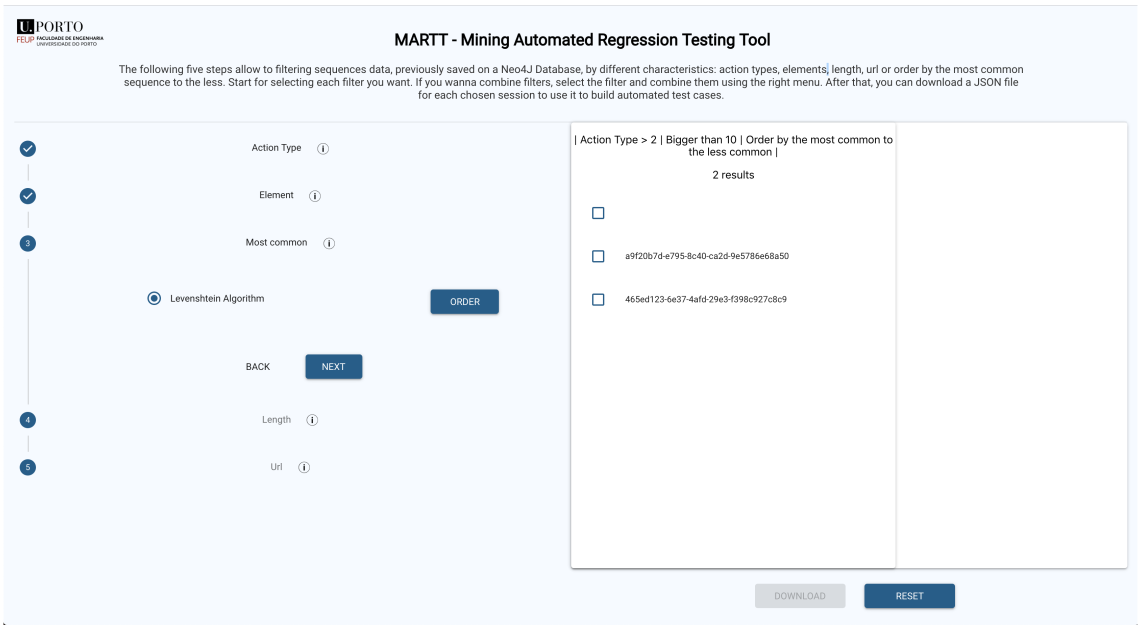
4899 results

- 0dce2697-d4dc-c229-0adb-d4818fe2485d
- c0b8b4da-468b-0cdc-8796-024732912e60
- a281f3fa-1237-009d-16b4-ba95c63f0f0e
- 2a2bfe25-7bc0-2db3-8bf4-17c09ed9880a
- b5c13352-7227-977a-5c6f-4ecf482e818a
- c8c8ad4a-b7c7-469c-5e3e-066eb959e4d4
- d0ff41bc-32f0-01e8-9921-5fd6ce4cdb63
- 0dfcc1a7-c690-a320-c3cc-47c617cc44ee

Active Filters

- Action Type > 2 X
- Bigger than 10 X
- Order by the most common to the less common X

Figure A.4: UI application with *Most Common* filter active.


MARTT - Mining Automated Regression Testing Tool

The following five steps allow to filtering sequences data, previously saved on a Neo4J Database, by different characteristics: action types, elements, length, url or order by the most common sequence to the less. Start for selecting each filter you want. If you wanna combine filters, select the filter and combine them using the right menu. After that, you can download a JSON file for each chosen session to use it to build automated test cases.

| Action Type > 2 | Bigger than 10 | Order by the most common to the less common |

2 results

- a9f20b7d-e795-8c40-ca2d-9e5786e68a50
- 465ed123-6e37-4afd-29e3-f398c927c8c9

Figure A.5: UI application combining three filters.

Appendix B

Test Case Example

A test case from our case study is represented in Listing B.1, as it was downloaded from our analysis application, in JSON format.

```
[{"path":"id(\"block-block-6\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/table[@class=\"mpricipal\"]/tbody[1]/tr[4]/td[@class=\"mtescolas lfnd\"]/a[@class=\"escola mtestg\"]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6647,"elementPos":9,"url":"http://www.ipvc.pt/esce-conferencia-mosquito-2019"},
{"path":"id(\"caixa-pesquisa\")","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6650,"elementPos":10,"url":"http://www.ipvc.pt/ese-prova-portugues-mestrados-profissionais-2018-19-agenda"},
{"path":"id(\"caixa-pesquisa\")/input[@class=\"search_texto\"]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6651,"elementPos":11,"url":"http://www.ipvc.pt/ese-prova-portugues-mestrados-profissionais-2018-19-agenda"},
{"path":"id(\"block-block-4\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/table[@class=\"mpricipal\"]/tbody[1]/tr[1]/td[@class=\"link investi mpover\"]/a[1]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":461,"elementPos":1,"url":"http://www.ipvc.pt/"},
{"path":"id(\"block-block-4\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/table[@class=\"mpricipal\"]/tbody[1]/tr[1]/td[@class=\"link inicio mpover\"]/a[1]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6638,"elementPos":2,"url":"http://www.ipvc.pt/IDi"},
{"path":"id(\"caixa-pesquisa\")","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":4951,"elementPos":3,"url":"http://www.ipvc.pt/"},
{"path":"id(\"caixa-pesquisa\")","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":4,"action":"dblclick","pathId":6639,"elementPos":4,"url":"http://www.ipvc.pt/"},
{"path":"id(\"caixa-pesquisa\")","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":3,"action":"input","pathId":6640,"elementPos":5,"value":["char","char","char","char","char","char"],"url":"http://www.ipvc.pt/"},
{"path":"id(\"block-block-4\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/table[@class=\"mpricipal\"]/tbody[1]/tr[1]/td[@class=\"link instit mpover\"]","
```

```
    session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","
    pathId":504,"elementPos":6,"url":"http://www.ipvc.pt/"},
{"path":"id(\"block-block-4\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/
    table[@class=\"mpricipal\"]/tbody[1]/tr[1]/td[@class=\"link instit mpover\"]","
    session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","
    pathId":504,"elementPos":7,"url":"http://www.ipvc.pt/"},
{"path":"id(\"block-block-6\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/
    table[@class=\"mpricipal\"]/tbody[1]/tr[4]/td[@class=\"mtescolas lfnd\"]/a[
    @class=\"escola mtesce\"]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","
    actionId":1,"action":"click","pathId":220,"elementPos":8,"url":"http://www.ipvc
    .pt/"},
{"path":"id(\"block-system-main\")/div[@class=\"block-inner\"]/div[@class=\"content
    \"]/div[@class=\"view view-pesquisas view-id-pesquisas view-display-id-page_1
    view-dom-id-4c67b94b909e90ad3d939d4c814c995c\"]/div[@class=\"view-content\"]/
    div[@class=\"views-row views-row-17 views-row-odd\"]/div[@class=\"views-field
    views-field-title\"]/span[@class=\"field-content\"]/a[1]","session":"a9f20b7d-
    e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6658,"
    elementPos":20,"url":"http://www.ipvc.pt/pesquisa"},
{"path":"id(\"block-system-main\")/div[@class=\"block-inner\"]/div[@class=\"content
    \"]/div[@class=\"view view-pesquisas view-id-pesquisas view-display-id-page_1
    view-dom-id-4c67b94b909e90ad3d939d4c814c995c\"]/div[@class=\"view-content\"]/
    div[@class=\"views-row views-row-17 views-row-odd\"]/div[@class=\"views-field
    views-field-title\"]/span[@class=\"field-content\"]/a[1]","session":"a9f20b7d-
    e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6658,"
    elementPos":21,"url":"http://www.ipvc.pt/pesquisa"},
{"path":"id(\"node-6897\")/div[@class=\"node-inner\"]/div[@class=\"content\"]/div[
    @class=\"field field-name-field-ncorpo field-type-text-long field-label-hidden
    \"]/div[@class=\"field-items\"]/div[@class=\"field-item even\"]/p[2]/a[1]","
    session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","
    pathId":6662,"elementPos":22,"url":"http://www.ipvc.pt/candidaturas-mestrados
    -2017-2018-2-fase-seriacao"},
{"path":"id(\"block-block-6\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/
    table[@class=\"mpricipal\"]/tbody[1]/tr[4]/td[@class=\"mtescolas lfnd\"]/a[
    @class=\"escola mtesce\"]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","
    actionId":1,"action":"click","pathId":6665,"elementPos":23,"url":"http://www.
    ipvc.pt/candidaturas-seriacao-2-fase"},
{"path":"id(\"caixa-pesquisa\")/input[@class=\"submeter\"]","session":"a9f20b7d-
    e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6652,"
    elementPos":12,"url":"http://www.ipvc.pt/ese-prova-portugues-mestrados-
    profissionais-2018-19-agenda"},
{"path":"id(\"block-system-main\")/div[@class=\"block-inner\"]/div[@class=\"content
    \"]/div[@class=\"view view-pesquisas view-id-pesquisas view-display-id-page_1
    view-dom-id-de6cf30471363511d668def84edce626\"]/div[@class=\"view-content\"]/
    div[@class=\"views-row views-row-1 views-row-odd views-row-first\"]/div[@class
    =\"views-field views-field-title\"]/span[@class=\"field-content\"]/a[1]","
    session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","
    pathId":6653,"elementPos":13,"url":"http://www.ipvc.pt/pesquisa"},
{"path":"id(\"block-views-lateral-block_3\")/div[@class=\"block-inner\"]/div[@class
    =\"content\"]/div[@class=\"view view-lateral view-id-lateral view-display-id-
block_3 view-dom-id-002c41d2a4debc11fa4847b0ac84cffb\"]/div[@class=\"view-content\"]/div[@class=\"views-row views-row-1 views-row-odd views-row-first views-row-last\"]/div[@class=\"views-field views-field-field-nimgtopo\"]/div[@class=\"field-content\"]/img[1]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6655,"elementPos":14,"url":"http://www.ipvc.pt/ese-prova-portugues-mestrados-profissionais-2018-19"},
{"path":"id(\"block-block-4\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/table[@class=\"mpricipal\"]/tbody[1]/tr[1]/td[@class=\"link inicio mpover\"]/a[1]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":2768,"elementPos":15,"url":"http://www.ipvc.pt/ese-prova-portugues-mestrados-profissionais-2018-19"},
{"path":"id(\"caixa-pesquisa\")/input[@class=\"search_texto\"]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":4,"elementPos":16,"url":"http://www.ipvc.pt/"},
{"path":"id(\"caixa-pesquisa\")/input[@class=\"search_texto\"]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":3,"action":"input","pathId":5,"elementPos":17,"value":["char","char","char","char","char","char","char","char","char","Enter"],"url":"http://www.ipvc.pt/"},
{"path":"id(\"caixa-pesquisa\")/input[@class=\"submeter\"]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6,"elementPos":18,"url":"http://www.ipvc.pt/"},
{"path":"id(\"block-system-main\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/div[@class=\"view view-pesquisas view-id-pesquisas view-display-id-page_1 view-dom-id-4c67b94b909e90ad3d939d4c814c995c\"]/div[@class=\"view-content\"]/div[@class=\"views-row views-row-6 views-row-even\"]/div[@class=\"views-field views-field-title\"]/span[@class=\"field-content\"]/a[1]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6658,"elementPos":19,"url":"http://www.ipvc.pt/pesquisa"},
{"path":"id(\"block-block-4\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/table[@class=\"mpricipal\"]/tbody[1]/tr[1]/td[@class=\"link instit\"]/a[1]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6667,"elementPos":24,"url":"http://www.ipvc.pt/candidaturas-seriacao-2-fase"},
{"path":"id(\"block-block-4\")/div[@class=\"block-inner\"]/div[@class=\"content\"]/table[@class=\"mpricipal\"]/tbody[1]/tr[1]/td[@class=\"link investi mpover\"]/a[1]","session":"a9f20b7d-e795-8c40-ca2d-9e5786e68a50","actionId":1,"action":"click","pathId":6668,"elementPos":25,"url":"http://www.ipvc.pt/candidaturas-seriacao-2-fase"}]
```

Listing B.1: Test Case Example.

Appendix C

Installation Manual

This work is mainly composed of two parts. The first part, regarding collecting data, needs the collecting script (*tracking.js*) to be run on the client side and the API responsible to save the collected data on the Database. Then, the analysis part relies on the API which connects to the Database, retrieving the needed data, and on the UI application, which allows the user/tester to define the constraints and filter the test cases wanted.

In the following steps, it will be described how to install and use these applications either locally, to be installed from scratch, or to use the version already installed on the currently used server.

C.1 Server

The server used to host this work was provided by the Software Engineering Laboratory, in the Faculty of Engineering of the University of Porto. To access the server, the following steps are required:

- Install VNC Viewer to access to the virtual machine.
- Connect to FEUP network (presential or by VPN).
- The virtual machine is running on the IP: `10.227.107.154` and port: 5900.
- Insert the credentials (they will be provided).

C.2 Database

C.2.1 Locally

- Install Neo4j desktop application [1].
- Create a new Graph database on the previously installed application C.1.

[1]https://neo4j.com/download-center/

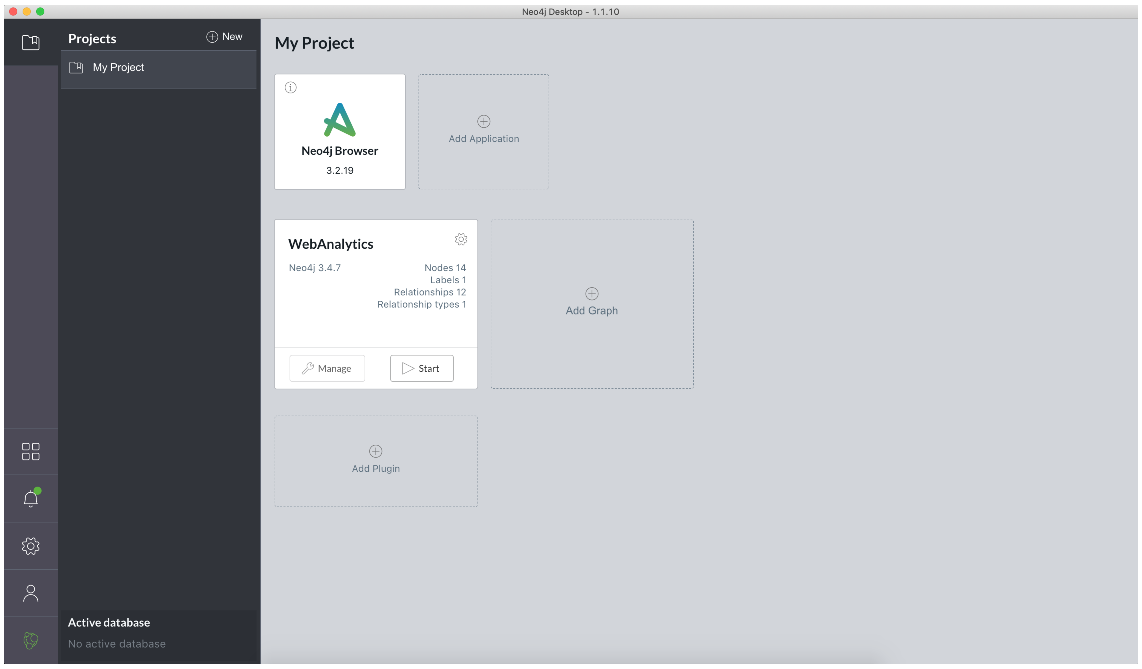


Figure C.1: Start Neo4j database in its desktop application.

- Start the database.

C.2.2 Server

- Install Neo4j: `sudo apt-get install neo4j=3.1.4`
- Run `sudo service neo4j restart`.
- Access the Neo4j interface at `http://localhost:7474/browser/`.
- To access by IP address, open config file by running

  `sudo gedit /etc/neo4j/neo4j.conf`
- Edit as the following example C.1

```
#dbms.connector.http.address
dbms.connector.http.listen_address = 10.227.107.15:7474
dbms.connector.bolt.listen_address = 0.0.0.0:7687
```

Listing C.1: Neo4j config file.

C.3 Collecting Data

C.3.1 Script: client-side

The JavaScript file must be included in the website to be tested, inserting the reference to the script in the HTML file. Currently, the script is hosted at `paginas.fe.up.pt` domain, in personal area. The link to access the file is the one in C.2.

```
<script type="text/javascript" src="https://paginas.fe.up.pt/~up201708898/tracking.
    js"></script>
```

Listing C.2: How to include Javascript file in HTML.

C.3.2 API: Save data

C.3.2.1 Locally

- Open app.js file.
- Choose port where API will be running (`app.listen(80)`)
- Change database credentials and the IP to the one where the Neo4j database is hosted (`neo4j.driver`)
- Run the `node app` command on the project's directory to start the API.

C.3.2.2 Server

- Run API by running the `sudo node app` command on the project's directory (currently `/Desktop/collect-data-api`)
- The API can be externally accessed at `http://web-analytics.fe.up.pt` when running on the port 80.

C.4 Data Analysis

C.4.1 Data Analysis UI

C.4.1.1 Locally/Server

- Run `npm install` on the first time application is running.
- In case of having trouble running the application, delete node_modules folder and run `npm install` again.
- Run `npm start` on the project's directory to start the application.

- The application should start on the port 3000, it should be accessible by the browser.

C.4.2 API: Retrieve data

C.4.2.1 Locally

- Open app.js file.
- Choose port where API will be running (`app.listen(3001)`)
- Change database credentials and the IP to the one where the Neo4j database is hosted (`neo4j.driver`)
- Run the `node app` command on the project's directory to start the API.

C.4.2.2 Server

- Run API by running the `sudo node app` command on the project's directory (currently `/Desktop/collect-data-api`)
- The API cannot be externally accessed, it requires VPN access.