CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Case Study on Test Optimisation and Visualisation of Diversity Information

Master's thesis in Computer Science and Engineering

Amar Kulaglić

Jonathan Helsing

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

A Case Study on Test Optimisation and Visualisation of Diversity Information

Amar Kulaglić
Jonathan Helsing

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

A Case Study on Test Optimisation and Visualisation of Diversity Information
Amar Kulaglić and Jonathan Helsing

Supervisor: Francisco Gomes de Oliveira Neto, Computer Science and Engineering
Advisor: Tobias Olsson, Volvo Car Corporation
Examiner: Jan-Philipp Steghöfer, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

A Case Study on Test Optimisation and Visualisation of Diversity Information
Amar Kulaglić
Jonathan Helsing
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Continuous Integration (CI) pipelines are vital in the implementation of CI and in the feedback cycles that surround automated testing in CI environments. A feedback cycle represents the time it takes from performing a commit until test results are ready. A significant problem in CI and automated testing is the long feedback cycles that come due to the increasing size of the test repository when executing test suites. The increased length of test execution is what this thesis will address using test case prioritisation. Through a design science methodology, we developed a tool and evaluated it by performing a case study at Volvo Car Corporation. The case study consists of two parts: evaluating the visualisation of data usually hidden during prioritisation, and data gathering and statistical analysis related to the performance of different distance measures and test case data. We have identified that similarity maps and history plots are good visualisation to enhance test decision making and maintaining and improving test repositories. Moreover, we have discovered the potential of using previous executions of test cases to determine their similarity.

Keywords: prioritisation, diversity, dimensionality reduction, continuous integration, similarity.

Acknowledgements

We begin by thanking all the people we have met along the way and made this thesis possible. Special thanks to Francisco for putting up with our questions and guiding us through this adventure. We would also like to extend a special thanks to Tobias Olsson and Johannes Reesalu at Volvo Car Corporation for providing support for everything from equipment to proofreading and to help us connect with the right people during our time there.

Last but by no means least, we would like to thank our friends and families for motivating and supporting us, but also putting up with us.

Amar Kulaglić and Jonathan Helsing, Gothenburg, June 2019

Contents

List of Figures

List of Tables

Abbreviations

APFD Average Percentage of Faults Detected. 2

CD Continuous Delivery. 8
CI Continuous Integration. 1

DBTCS Diversity-Based Test Case Selection. 9

ECU Electronic Control Unit. 10

MDS MultiDimensional Scaling. 14

NCD Normalized Compression Distance. 11

PCA Principal Components Analysis. 14

SNE Stochastic Neighbor Embedding. 14
SUT System Under Test. 3

t-SNE t-Distributed Stochastic Neighbour Embedding. 15

VCS Version Control System. 1

1

Introduction

Continuous Integration (CI) is widely used by practitioners to merge and automatically test each developers' change or addition of software code to a baseline. This baseline is shared among a team consisting of, e.g., testers, developers, and architects. Performing automated testing determines whether a merge can result in a success or an integration failure. However, when using automated testing with various regression testing methods, and on large software systems, the execution of such a suite can take several hours or even days [1], [2]. These long execution times become prohibitive in practice where developers want rapid feedback from running tests so that they can identify (i.e., debug) and correct issues as soon as possible. Current practice can result in hours of waiting for test results to know if even a small code change succeeded or not. One way to shorten the time to receiving result is to use test optimisation techniques.

There has been much research into different test optimisation methods, which are divided into three types: test case selection, test case prioritisation and test suite reduction [3]. The focus of this study is on test case prioritisation. The goal with prioritisation is to assign specific order (i.e., priorities) and determine which test should execute first, second, third and so on. Before it is possible to use any prioritisation technique, the test case data (e.g., failure rate and test case names) has to be formatted into something that the prioritisation technique can understand. This is referred to as different encoding strategies. With the properly formatted data, the prioritisation algorithm can assign a specific order to the test cases. There are two types of information that can be formatted. The *static* type of information refers to all information that does not require the software to run. For example, test specifications, test case code and requirements are all examples of static information. Conversely, we consider pass and failure information to be of the *dynamic* type. Therefore, one of our goals is to look at the different trade-offs in using different types of information for prioritisation. Moreover, we will look at how using the information created during test case prioritisation for visualisation can help developers and testers to refine and maintain the quality of their test repositories. A test repository is a location where all tests associated with a project are stored. The test repository is often located in a Version Control System (VCS).

Maintaining a test repository means creating, removing and changing test cases and plays a vital role in test cycles. With test cycles, we mean the steps taken to define

tests, set-up and maintain test environments, execute tests and analyse execution results, including maintaining the test repository itself. This process repeats until testing efforts on the project end. A connected term is feedback cycles, which relates to the time that has passed since test execution is triggered until developers receive the execution results.

In addition to having different type of information, test artefacts also have different levels, e.g., unit and integration level tests. Each test level corresponds to a specific purpose of the test while testing different aspects of a system, and this study explores integration testing and its automation. The purpose of integration testing is to test if various software modules work as expected when they are combined.

Test optimisation techniques, including test case prioritisation, are often costly to use; as such, it is of vital importance to know the effectiveness of such techniques. A well-known measurement of the effectiveness of test suites to detect faults is the Average Percentage of Faults Detected (APFD) value [4], [5]. This measure says how fast a test suite detects the different faults in the code. However, it does not consider how fast the optimisation technique itself is and is thus commonly used to measure the effectiveness of prioritisation techniques. We can capture the efficiency in terms of time by comparing the techniques with each other based on the time required to prioritise tests and execute them.

1.1 Statement of the problem

The main problem addressed by this thesis is that while current CI practices demand faster feedback cycles, test cycles become longer and more expensive to run. In Section 1.1.1, we will present details regarding long test feedback cycles as a problem in industry and how to solve this problem. The solution will in turn lead to another problem caused by the black box of test case prioritisation, which will be shown in Section 1.1.2. Also, a solution for this problem will be presented in the same section.

1.1.1 Long test feedback cycles

Nowadays, automated testing is widely used in industry, and feedback cycles from smaller test repositories are usually manageable when executing tests. However, this is necessarily not the case for companies with larger test repositories since their feedback cycles can become too long to be useful for their developers. The reason behind this is that the time needed to run the entire test repository increases as the size of the repository increases.

Companies may in some situations increase their resources spent on testing to compensate for longer feedback cycles. However, this is only a short term solution, and therefore the longer feedback cycles can return sooner or later, resulting in the pos-

sibility of a low return on investment. In some cases, it may not help by investing more resources in testing since some tests may depend on time for completion. An example of such a case is a test evaluating if an operating system automatically enters sleep mode after a certain amount of time. Thus the test requires the operating system to wait until this time has passed until it can evaluate if it succeeded or not.

A solution for the long term is to use test prioritisation based on diversity to decrease feedback cycles. In the context of test optimisation, the diversity of test cases refers to how different they are based on some test case data. Previous research has shown the potential of diversity based test optimisation [4], [6]–[8]. A drawback with diversity based prioritisation is that they often are costly to use [4], which is one of the reasons why we have not seen large-scale adoption of automated test prioritisation based on diversity in the industry.

1.1.2 The black box of test case prioritisation

Automated test prioritisation using diversity, provides information (e.g., failure rate and diversity) that is kept under the hood and is embedded in the technique itself. This will lead to testers missing a lot of information, resulting in them having no idea how the optimisation of the automated technique is happening since they see it as a black box. The information from the automated technique can be useful for the testers if combined with their expertise when making testing decisions. Moreover, it can be used to identify patterns in test executions that can lead to insightful discoveries about the System Under Test (SUT) or the development process itself.

There is a solution to the problem mentioned above, which is about using dimensionality reduction for visualising diversity information of different test cases from the test repository in a way that is interpretable for testers. The dimensionality reduction reduce the number of dimensions found in diversity information, which is a byproduct from the result of test case prioritisation. The results of the dimensionality reduction can be used to visualise information about the test repository, which becomes valuable for developers and testers in their efforts to maintain and improve the quality of the test repository [9]. The potential of using the diversity information in the way presented here, the total cost of diversity based test optimisation might become more manageable with the additional information made available to testers, developers and managers. Thus increasing the value gained from investing in diversity-based test prioritisation.

1.2 Purpose of the study

Our aim is to use design science to deepen our knowledge and understanding of how test case prioritisation can be used in CI pipelines and what effects it can provide. This will be achieved by providing tool support for a technique to use different types

of information to prioritise test cases in a CI pipeline. The tool will also support visualising information used in prioritisation. Thus, we contribute to the technical knowledge about developing a test prioritisation and visualisation tool. Moreover, it will contribute to the scientific knowledge about how test case prioritisation based on static and dynamic information can complement each other along with test case visualisation in a CI environment.

1.3 Research Questions

This study aims to answer three main research questions. The first question (RQ1) is about how test optimisation tools can be used in CI pipelines. Meanwhile, the second research question (RQ2) relates to how information from test case prioritisation can be visualised to support humans in decision making. Additionally, the last research question (RQ3) relates to the effects of using test optimisation in CI environments. Examples of related practices are constant testing and code merging, and examples of technical solutions are version control systems, software repositories and toolchains.

RQ1 How can we instrument CI pipelines to optimise test feedback cycles?
RQ1.1 What different types of information can we use to optimise tests in CI pipelines?
RQ1.2 What are the trade-off in using different types of information from test artefacts?
RQ1.3 How effective are the optimised test suites?

RQ2 How can we use test optimisation to support human decision making in test cycles?
RQ2.1 What type of information do stakeholders use to make testing decisions?
RQ2.2 To what extent can we capture and visualise this information to trigger insights from testing cycles?

RQ3 How does the integration of test optimisation in CI pipelines affect practitioner's feedback cycles?

The main contributions and findings of this thesis are summarised below:

- Stakeholders use two types of information to make decisions in testing: administrative and testing information. The first type is related to experience and situational awareness. On the other hand, testing information relates to the knowledge of testing efforts (such as test cases and execution data).

- Dynamic information have much higher fault coverage than their static counterpart. On the other hand, the static type perform better when it comes to feature coverage. This outcome is not surprising, but it will lead to promising

future work.

- Similarity maps based on t-SNE provides more insights among practitioners than those based on MDS.
- The construction of a heat map visualising execution results of test cases over time. This visualisation has shown great potential in producing insights among practitioners on the status of both the testing environment and test environments.

Chapter 2 of this thesis presents the background and related work. The chapter presents details used in prioritisation and the algorithms used in dimensionality reduction. The chapter that follows, Chapter 3, contains the design-science methodology used in this thesis, which details the tool's development and evaluation of the tool as a case study. Chapter 4 presents the developed tool, including how it operates and its architecture. Chapter 5 details the results of its evaluation. Chapter 6 presents a discussion on our results, threats to validity. The chapter finishes by presenting future work and our conclusion.

2

Background and related work

Testing is an essential part of quality assurance, verification and validation of software systems [3]. Therefore, automated testing is an integral part of CI pipelines, where tests execute when there are code base changes, to ensure that both integration of different parts of the software work and that the software upholds its quality and validity. As such, a CI pipeline often contains both a VCS and a build system which compiles the code and runs the tests. The result from tests is feedback for developers who use them to make decisions on the next step in the software's development cycle, as well as knowing if a solution to a problem worked or not.

The basic principle of CI is to build the software at every change [10], and the reasoning behind CI is further introduced in Section 2.1.

The purpose of test case prioritisation techniques is to sort test cases in an order that maximises, or minimises, an objective function, e.g. rate of fault detection [5]. A significant advantage of this technique is if test execution is interrupted (e.g. due to timeout constraints, or limited hardware availability), developers and testers can be sure that the most significant test cases were executed first.

Prioritisation can be used in black-box or white-box testing. If tests are done using black-box testing, the test optimisation is independent of the software's source code, e.g. static analysis. The opposite, white-box testing, means that test optimisation algorithms use the information that is available during run-time and knowledge of the source code. Black-box information includes requirements, test case names, test case code, execution result and fault or failure detection. Similar examples of white-box information is source code coverage and function or method coverage. In black-box testing techniques, diversity-based are among the two best performing techniques, together with combinatorial integration testing [8]. Moreover, many studies present the benefits of using diversity in test optimisation [4], [6]–[8]. Thus, the use of diversity-based prioritisation is justified for determining the effects of the two types of information. Details of diversity-based optimisation is presented in Section 2.2.

Visualising the information used in test prioritisation requires the use of dimensionality reduction.

2.1 Continuous integration

CI is the practice of frequently compiling software, executing automated tests and inspections, deploying software, and receiving feedback [10]. It is a practice which tries to answer questions that usually are brought up early by developers. For example, "Are we following coding standards?" or "What is our code coverage?". If they apply CI, they will have answers to these, and many more questions each time a change appears in the VCS.

A major problem in software engineering is assumptions [10]. For example, by assuming that developers follow coding and design standards, the resulting software will most likely be difficult to manage. Each assumption made increases the risks of the project. However, many assumptions can become known facts if we use CI [10]. The example assumption above is mitigated by using CI with automated software inspection.

Paul M. Duvall *et al.* [10] presents the high-level values of CI to be to reduce risks and repetitive manual processes, generate software that is deployable anywhere and at any time, provide the project with better visibility, and finally, to give the development team greater confidence in the software.

There are many different mechanics to CI: Continuous Database Integration, Testing, Inspection, Deployment, and Feedback [10]. As also noted by Duvall *et al.* [10], there is often a discretion between the software and related databases and, as such, it is good to rebuild the database continuously. This rebuilding means the software and database are synchronised. Continuous Inspection enables teams to reduce code complexity and the amount of duplicated code, but also to maintain code standards [10]. At every integration, the code is inspected by the build server to asses how it conforms to specified rules. Continuous Delivery (CD) is another practice, and it can be most comfortable described as CI but for deploying the software to production systems. Humble and Farley [11] motives the need for CD to replace the manual process often employed by organisations with a process that frequently delivers the product to production systems.

Continuous testing is the practice of automatically running tests for each change in the VCS. It is an essential aspect of CI since it allows developers to evaluate the software. However, since Duvall *et al.* [10] states that developers should wait until their integration has succeeded before starting on another task, feedback time becomes essential. If the automated tests take a long time to complete, the development will slowly grind to a halt due to developers waiting for their commits in the VCS to pass. As software projects grow in size, the ability to perform test prioritisation becomes more important.

2.2 Diversity-based test case prioritisation

Diversity-based techniques build on the assumption that diverse test suites perform better than test suites containing very similar test cases [6], [7]. Diversity can be expressed in many different ways, e.g., the diversity of requirements coverage and the diversity of experience among developers. However, in the context of test prioritisation it refers to the difference between test cases with regards to some specific test case data. This difference is expressed as the distance between test cases. In this thesis, diversity does not capture semantic differences, instead it goes into the lexicographic differences [1].

A technique presented by Cartaxo *et al.* [3], Diversity-Based Test Case Selection (DBTCS) uses the diversity between all test cases to select those that result in the most diverse test suite. Their technique is easily modified to perform prioritisation instead of selection resulting in the process containing the three following steps, as seen in Figure 2.1: Encoding, diversity calculation and prioritisation.

Next, we will explain the terminology and existing strategies that implement DBTCS. We illustrate the technique in Chapter 4, when explaining our tool contribution.

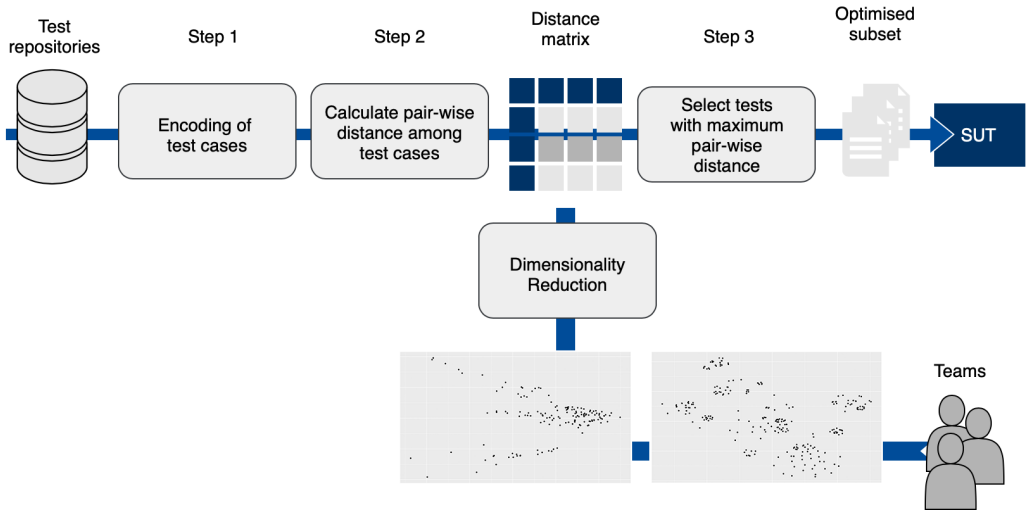


Figure 2.1: The steps generally taken in diversity based test optimisation. Including the resulting similarity maps presented to the teams. This figure is based on a figure presented by de Oliveira Neto *et al.* [9].

2.2.1 Encoding strategies

Encoding strategies prepare the information that the distance measures will use by placing it in a vector and ensuring that this data is in a format that the measures can use. Given the many properties of test cases (including code, name and

documentation), an example of an encoding strategy on their names is to extract the name from the test case and place these names in a vector that the distance measures can use. If we also extend this example by applying a naming convention for test cases that test different Electronic Control Unit (ECU), the convention can look something like this: *ECU-name_feature-name*. Here, an encoding strategy is to take the test case names and then remove the ECU portion of the names, before placing them in a vector. By having multiple encoding strategies, techniques can use string matching to calculate the similarity of tests based on the name of the features or the combination of features and ECUs. This functionality becomes useful when there are many similar test cases that perform the same test on different ECUs and executing all of them could be wasteful. Instead, we would like to run our tests on a diverse set of features distributed across a mix of ECUs[1].

There are two types of test information that can be encoded, static and dynamic information. The static type contains information that do not require running the SUT. For example, test case names and their code are examples of static information, which is available without running the software they test. On the other hand, test data of the dynamic type, require executing the test for the information to be available. An example of such information is the execution history of a test case, which is the previous results from running the test case.

Every organisation conducts testing differently. Thus strategies for encoding vary as well. For example, company A may use specific guidelines for naming test cases where the tested feature is a part of such names. Meanwhile, company B does not employ any guidelines regarding naming tests, resulting in each test case having different naming structures. If company A extracts and prioritises test cases based on which feature the test evaluates, then their encoding strategy for acquiring the feature from the test case name only works at that company. If company B uses the encoding strategy present at company A, they might not even be able to prioritise any test cases.

It is not only naming guidelines for test cases that are unique for different organisations, but it is also the structure of test cases into suites. Since all organisations have their guidelines and ways of encoding test cases, we are not able to directly take the tests from one company, and use it in another company in a different domain. For example, one test case might test if it is possible to turn off the radio in a car; it is not possible to execute the same test case on a social media platform.

In short, when interested in prioritising diverse tests, a company should first realise "diversity in terms of what?". Literature explores different options, such as using test artefacts (static information), or execution history (dynamic information). Each type of information yields different test case data, and here we select a subset from each to study.

[1]Note that different ECUs can cover similar features, and similarly, similar features can be covered by different ECUs

2.2.2 Distance measures

Distance measures, also known as distance functions, can calculate the distance between test cases. The measures use either sets or sequences, and the main difference is that set-based measures do not take the order into account [12]. Generally, distance measures are pair-wise, meaning that they only calculate the distance between a pair of inputs, for example, two vectors. The basis for these calculations is how much the two input data objects has in common. If they have nothing in common, the resulting distance is 1, and on the contrary, if they are identical, their distance is 0.

All distance values for each pair of entities are organised in a matrix referred as a Distance matrix. A distance matrix is a $n \times n$ matrix, where n, in our context, is the number of test cases in a test suite. Consider $T = \{t_1, t_2, ..., t_n\}$ to be a test suite and each t_k, for $k \in \{1, 2, ..., n\}$ is a test case in T. Then, each element $x_{i,j}$ in the distance matrix represents the distance between t_i and t_j [3]. We need to observe that the distance matrix is symmetric, meaning that the distance between each pair of test cases appears twice (i.e., one appearing in the upper triangle of the matrix and the other one in the lower triangle of the matrix). For example if we look at t_1 and t_2 as two test cases, the value of $x_{1,2}$ located in row 1 and column 2 in the matrix is equal to the value of $x_{2,1}$ located in row 2 and column 1 in the matrix. The distance matrix forms the basis of the prioritisation process described in the next section. One can use string distance algorithms [2] to calculate the distance x_{ij}, given that the information on test case i and j is a string. Often test cases are written using a sequence of instructions that can be represented using strings [2], this includes e.g. Robot Framework test cases, steps in user actions written in natural language and a scenario specified in a domain-specific language.

Both of the distances presented below, Jaccard Index and Levenshtein distance, are examples of string distance algorithms. Out of these two distance measures, Jaccard Index is set-based, while Levenshtein on the other hand, is a sequence-based distance measure. Moreover, the Normalized Compression Distance (NCD) is another example of a distance measure and can be considered a universal distance metric [6], [13].

Jaccard Index: The Jaccard Index, as introduced by Paul Jaccard [14], uses the relation between what exists in one entity and that of another. For example, given two test cases, A and B, their similarity can be calculated using the Jaccard Index as follows:

$$jaccardSimilarity(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The Jaccard Index formula, also known as the Jaccard similarity coefficient, is the size of what A and B has in common, i.e. their intersection, divided by their combined size, i.e. their union. As opposed to the similarity calculation above, the mathematical definition of Jaccard distance is defined below for the two test cases

A and B:

$$jaccardDistance(A, B) = 1 - jaccardSimilarity(A, B),$$

Levenshtein: The simplest explanation of the Levenshtein distance is that it calculates the smallest number of edit operations (deletions, insertions and replacements) that changes one of the input strings into the other [15]. Levenshteins original description details the use of this distance for correcting binary words, however, it is also possible to extend it to letters and words of other alphabets than $\{0, 1\}$. For example, say that there are two test cases, A and B. Test case A has the name *"Set Status Online"* and B *"Set Status Offline"*. The Levenshtein distance between these two test cases is 2 because, it takes 1 replacement, the *"n"* in *"Online"* to *"f"* and 1 insertion of an *"f"* after the replacement character.

Normalised Compression Distance: Although the earlier distance measures are string based, there exists a universal cognitive diversity distance called Information Distance [16]. Information distance uses the noncomputational Kolmogorov complexity [6], [7], [16]. The complexity, $K(X)$, is defined as the length of the shortest program that prints the binary string x and then stops [17]. Bennett *et al.* [6] define information distance as the length of the shortest program that translates the binary strings x and y into each other. However, even if Kolmogorov complexity is noncomputational, it is possible to approximate its value by using compressors as proven by Cilibrasi and Vitányi [13] when they introduced NCD. They present the NCD calculation using a compressor $C(X)$ and from the two binary strings x and y we get $NCD(x, y)$ as:

$$NCD(x, y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$

Here, $C(y)$ is the length of the binary string y after compression with the compressor C and $C(xy)$ is the length of the joint binary strings x and y after compression. Moreover, Cilibrasi and Vitányi presents the value of $NCD(x, y)$ as $0 \leq NCD(x, y) \leq 1 + \epsilon$ where ϵ is an error to the approximation of the compressor used. For example, they go on to present that the compression algorithms *gzip* and *bzip2* has an ϵ above 0, while the compressor *PPMZ* presented an ϵ equal to 0.

2.2.3 Prioritisation

Our prioritisation algorithm builds on a greedy selection algorithm as introduced by Cartaxo *et al.* [3]. The main idea behind greedy selection algorithms is that for each test case it selects, the test case selected will be the one with the highest distance

in the distance matrix. Thus it is relatively simple to extend the greedy selection algorithm to apply to prioritisation as well. This is done by forcing it to select the same number of test cases as there are in the original test suite. After our greedy prioritisation algorithm identified the highest distance in the distance matrix, it calculates the total sum of distances from each of the two test cases to all other test cases. In other words, we take one of the two test cases and summarise the distances between it and all other test cases in the matrix. This will be further detailed in Section 4.1.1. Later, the test case with the highest sum would be added to the prioritised suite and removed from the distance matrix. Our greedy prioritisation algorithm repeats this process until the distance matrix is empty.

2.3 Dimensionality reduction

Ensuring a diverse test repository is important to avoid wasteful test artefacts [9]. To support keeping a diverse test repository de Oliveira Neto *et al.* also suggests using Similarity Maps to support developers and testers in their efforts in maintaining test repositories and identifying issues with them. An example of a similarity map is found in Figure 2.2. The figure shows how a distance matrix suitable for test case prioritisation, also can be utilised to visually show similarity and distances between multiple test cases. When looking at this distance matrix, TC1 and TC2 is the most similar pair of test cases (excluding self-comparisons) with a distance of 0.333. The similarity map in Figure 2.2 also shows this by positioning them close to each other. The same principle goes for TC6 when comparing it to TC3 and TC4, where TC6 and TC3 with the distance 0.916 are more diverse compared to TC6 and TC4 with the distance 0.900. Thus, TC6 and TC3 are further away from each other than TC6 and TC4 in the similarity map. With those scenarios, an important characteristic is shown that the similarity map puts test cases close to each other when the distances between the test cases are small in the distance matrix. Same principle goes for distances that are large, but the test cases are instead positioned far apart from each other in the similarity map.

	TC1	TC2	TC3	TC4	TC5	TC6
TC1	0.000	0.333	0.916	0.900	0.888	0.571
TC2	0.333	0.000	0.916	0.900	0.888	0.571
TC3	0.916	0.916	0.000	0.923	0.916	0.916
TC4	0.900	0.900	0.923	0.000	0.900	0.900
TC5	0.888	0.888	0.916	0.900	0.000	0.571
TC6	0.571	0.571	0.916	0.900	0.571	0.000

TC1 = "a b c d"
TC2 = "a b c e"
TC3 = "h l j k l m n"
TC4 = "o p q r s"
TC5 = "w x y z"
TC6 = "a b y z"

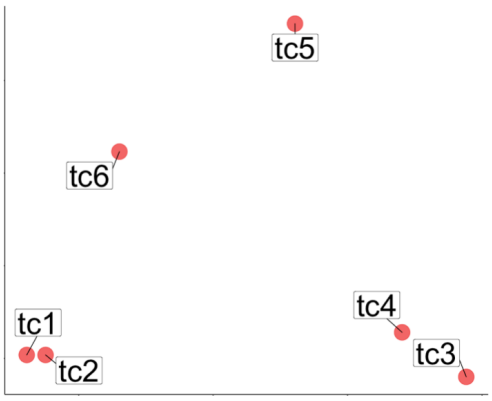


Figure 2.2: An example of a similarity map created using a custom set of test cases. Distance matrix created using Jaccard Index and similarity map was generated using t-SNE.

Even for such a small test suite as the one in Figure 2.2, the distance matrix becomes hard to interpret and digest due to it having too high number of dimensions. The number of dimensions is as high as six in the distance matrix since it inspects six test cases. Thus, the number of dimensions in a distance matrix is as many as the total number of test cases that are being inspected in the test repository. To be able to go from the distance matrix to the similarity map we need to use a dimensionality reduction techniques. These techniques takes a multidimensional vector and convert it to a lesser dimension. In our case, it takes a distance matrix and reduces the matrix to two dimension which we are able to plot, thus the visualisation of the distance matrix is something that will support testers and developers in maintaining their test repositories [9].

When it comes to dimensionality reduction techniques, there are two types: linear and non-linear. The difference between linear and non-linear techniques is that linear techniques focus on keeping dissimilar data points in the low-dimensional representations far apart, while for high dimensional data, non-linear techniques focus more on keeping similar data points in their low-dimensional representation close together [18]. Linear techniques include, among others, classical MultiDimensional Scaling (MDS) [19] and Principal Components Analysis (PCA) [20] [21], and they function by embedding the data in a linear subspace that has a lower dimensionality [22]. On the other hand, non-linear techniques include, among others, Stochastic

Neighbor Embedding (SNE) and t-Distributed Stochastic Neighbour Embedding (t-SNE) [18].

There are some pros and cons with linear and non-linear techniques. One drawback with using linear techniques is that they do not have the ability to deal with complex non-linear data, which the non-linear techniques have [22]. In this study, we will focus on MDS, that has shown promising results in earlier studies for visualising test data [9] and t-SNE which has been shown good results in different domains [18].

2.3.1 Multidimensional Scaling

MDS, as introduced by Torgerson [19], is a dimensionality reduction technique that visualises data points in a high-dimensional space to a low-dimensional space by taking all pairwise distances of each pair of data points as input so that these distances are preserved in the low-dimensional space [23]. There exist two types of MDS techniques, namely metric and non-metric, where they differ in how their calculations are performed and what metrics that are used by them [23]. Metric MDS is also called classical MDS, metric scaling [24] or classical scaling [22], and is a linear technique [22]. According to Sumithra V.S and Subu Surendran [23], non-metric MDS is a non-linear dimensionality technique. The difference between metric and non-metric MDS is that the latter one uses the former, but also performs additional steps afterwards [24]. In other words, Goodhill, Simmen and Willshaw mentioned in their report that non-metric MDS starts by using metric MDS to select a configuration of data points in the targeted low-dimensional space to get an ordering of distances. They continue and describe the next step of the non-metric process as comparing this ordering of distances with the ordering of dissimilarities from the input matrix and pinpointing inconsistencies. Moreover, they say that with these orderings, non-metric MDS tries to match them so that each distance achieves a target distance.

According to van der Maaten, Postma and van den Herik [22], there are two main weaknesses in metric MDS. They state that the first weakness is that there is a proportionality between the size of the covariance matrix and the dimensionality of the data points. However, this weakness could be avoided in some situations where the number of data points in a data set is less than the number of dimensions. The second weakness, as mentioned in the same report, is that large pairwise distances are primarily targeted in metric MDS, which is not as significant as targeting small pairwise distances.

2.3.2 t-Distributed Stochastic Neighbour Embedding

t-SNE, as introduced by van der Maaten and Hinton [18], is a non-linear dimensionality reduction technique based on machine learning that also projects data points in a high-dimensional space to a low-dimensional space of two or three dimensions as

in MDS that we mentioned in the previous section. The result is something that is interpretable and understandable for humans. van der Maaten and Hinton go on to state that the technique is giving each data point a position in the low-dimensional map so that it keeps similar data points close together. Moreover, they say that student-t distribution is used in t-SNE to calculate the similarity of all pairs of the data points in the low-dimensional space. t-SNE was invented to improve SNE by Hinton and Roweis [25] since some shortcomings were found, which van der Maaten and Hinton [18] stated in their report.

The shortcomings concerns optimisation problems with a cost function used in SNE and by another problem referred to as the *crowding problem*. According to van der Maaten and Hinton, the crowding problem is that the area of a two-dimensional map, which can accommodate moderately distanced data points, will not be large enough compared to the area required to accommodate data points that are nearby. Therefore, if small distances with high accuracy in the map wants to be modelled, the majority of the points that are moderately distant to a data point i will be placed too far away in the two-dimensional map [18]. However, van der Maaten and Hinton mentioned further in their report that the two problems in SNE were mitigated by making some adjustments to the cost function used in SNE so that t-SNE uses heavy-tailed distributions. They stated that the first adjustment is that the cost function in t-SNE uses Student-t distribution instead of the Gaussian distribution when calculating the similarity of two points in the low-dimensional space and that the second adjustment is about t-SNE featuring a symmetrised version of SNEs cost function that was introduced by Cook *et al.* [26].

When using t-SNE, there are some important parameters involved that control the optimisation of t-SNE and how visualisation of the data points will be presented in the low-dimensional map. These parameters are, among others, perplexity, early exaggeration, learning rate, number of iterations. The values of these parameters are chosen manually by the user of the technique (e.g., a researcher) and are different for different data sets, but there are some typical values for some parameters as presented in van der Maatens and Hintons report [18]. In their report, they defined perplexity as a measure of the effective number of neighbours, which has typical values between 5 and 50. By this, it means that smaller perplexity gives more abandoned and self-contained data points in the low-dimensional map. However, if the values for this parameter are too small or too high, it will result in unexpected behaviours that is not useful from the visualisation of t-SNE. Therefore, it is important to experiment with different values that suites the chosen data set. Further on, van der Maaten and Hinton stated also in their report that the parameter called early exaggeration affects the empty spaces between the natural clusters in the low-dimensional map, which in turn can help identifying a good global organisation.

Although the origin of t-SNE came from an improvement of SNE, it has some disadvantages. In van der Maatens and Hintons original report [18] about t-SNE, they stated that one weakness is that it is ambiguous regarding how the general dimensionality reduction tasks are completed in t-SNE. This applies to data whose dimensionality is reduced to strictly more than three dimensions. Another weakness

that they also mentioned is that t-SNE is easily affected by the *Curse of the Intrinsic Dimensionality* of the data since local properties of the data are used when performing the dimensionality reduction. The *Curse of Dimensionality* is defined as the number of variations that a function has after learning, which is more detailed in Bengio's report [27]. For example, if we have a data set with a high intrinsic dimensionality, t-SNE may break its own assumption about local linearity and therefore not be successful [18]. The third weakness, which is presented in van der Maatens and Hintons original report, is that the cost function in t-SNE is not convex. The outcome of this, is that some parameters regarding optimisation needs to be selected [18], which is not always desired.

3

Methodology

This thesis uses the design-science methodology [28], commonly found in the field of Information systems. The effect of this is that we develop a prototype tool, also known as a solution, which is then evaluated using a case study. The section that follows (Section 3.2) details the research questions. Section 3.1 shows specifics on the design-science methodology, i.e. the development and design of the tool. Meanwhile, Section 3.3 presents the methodology used in the case study evaluation. Chapter 4 displays the resulting tool, while the results of its evaluation, as a case study, is seen in Chapter 5.

3.1 Development

The basis of design-science is the iterative development of solutions to specific problems. The iterations are called cycles. Design-science research has to fulfil three criteria: The research has to be relevant, novel and use rigorous methods. Engström *et al.* [28] developed the *visual abstract template* which presents the definition of these three criteria, but also the problem and its solution. Therefore, Figure 3.1 shows the visual abstract for this design-science thesis.

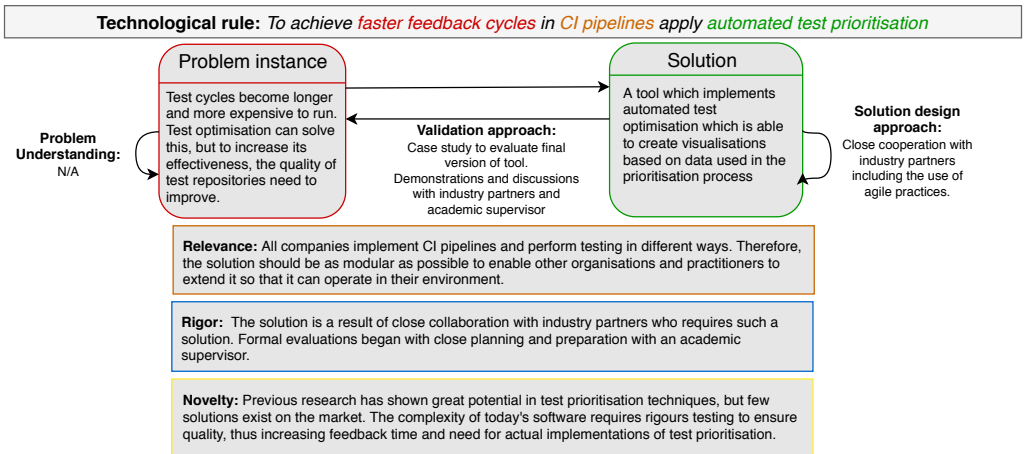


Figure 3.1: Our design-science visual abstract presented according to Engström *et al.* [28].

This study contains three cycles of the design-science research methodology introduced by Peffers *et al.* [29], and as seen in Figure 3.2. Each block in Figure 3.2 corresponds to a specific phase during design-science research. The *Identify & motivate problem* phase involves researching the problem to be able to position the study within the field. During the phase that follows, we define the goals and constraints of the tool, i.e. what it will be capable of and what it will not do. The *Design & Development* phase meant that we developed and designed the tool before demonstrating and evaluating it. After the first evaluation, we performed two more cycles before reaching the *Communication* phase, which resulted in this report.

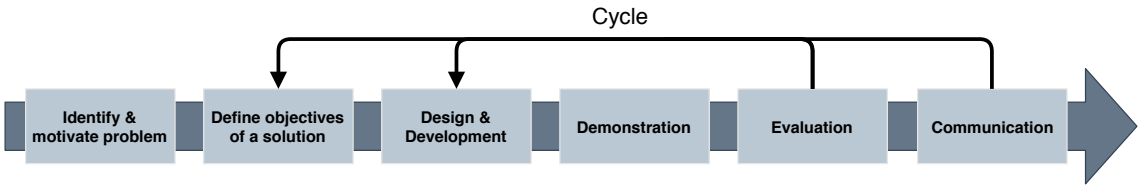


Figure 3.2: The phases of design-science research as adapted from Peffers *et al.* [29].

Below, we will discuss the three cycles that were a part of this study and describe the phases involved in each cycle with regards to what we did during them.

First Cycle

- *Identify & motivate problem* - The problem addressed by this thesis was identified and motivated through meetings and discussions with the academic supervisor and industry partners.
- *Define objectives of a solution* - Through discussions with our industry partners and analysis of their CI infrastructure, we established an outline of a potential solution together with our academic supervisor.
- *Design & Development* - The result of development and design was a tool with baseline graphical support for visualisation and the first set of associated libraries. Moreover, we had the basics in place for calculating diversities of strings and parsing test cases.
- *Demonstration* - The prototype tool was showcased to both academic supervisor and the industry partner.
- *Evaluation* - We evaluated the tool through discussions with industry partners and the academic supervisor. The industry partners presented changes in their needs, while academia brought forward what additions was needed to be able to perform the study. Since our tools was developed in close collaboration with our industry partners, we continuously received feedback on the tool itself, thus reducing the need of full-scale evaluations for each cycle.

Since tool development was performed with a close connection to the industry partners, we can validate the tool in this way. This relationship reduced the need for a large-scale evaluation of the tool at the end of each cycle.

Second Cycle

- *Define objectives of a solution* - The focus here was to redefine the objectives for the tool to better match changes to the stakeholders' needs, with regards to both industry partners and academia.

- *Design & Development* - The results of this development cycle was a tool that able to produce better visualisations since we changed which libraries were used. Moreover, the tool was able to perform prioritisation using execution history and test case names.

- *Demonstration* - The demonstration was a half-time showcase highlighting the changes made from the previous demonstration. Again, the tool was shown off to both our industry partner and the academic supervisor.

- *Evaluation* - The evaluation in the second cycle was very similar to the first cycle. However, on our part, the needs would be affected by what we would be able to implement in the given time frame.

Third Cycle

- *Design & Development* - The outcome from development in this final cycle was fine-tuned visualisations (similarity maps and history plot) and scripts for calculating different measures, such as fault and feature coverage, as well as APFD.

- *Demonstration* - Presentation at the industrial partner.

- *Evaluation* - We performed a case study to evaluate the tool and its output. The methodology used for this case study can be found below (Section 3.3). The outcome of the evaluation can be found in Chapter 5.

- *Communication* - The study was communicated through a thesis report examined by both academia and industry stakeholders. Moreover, a presentation were held with industry stakeholders showcasing its use and its architecture. Additionally, discussion were held with an industry stakeholder who was able to further the development and use of the tool at the case company.

The tool uses a modular design so that it easily can be extended to work in different systems and environments. The tool was programmed using Python 3[1] to support the different systems in use by the industry partner. We adopted some practices from Scrum and agile techniques to help us track what was to be implemented and to support problem-solving. We used a Scrum board to keep track of our

[1]Python 3 - https://www.python.org

backlog of new features waiting for implementation. Moreover, the board tracked priorities, statuses, and who was working on what feature. At the start of every week, we discussed what features we should implement, whom we might need to contact regarding questions and the overall status on deadlines. From time to time, we employed pair-programming to improve the quality of the code and knowledge sharing.

During the development process, well-known libraries were used to ensure a higher quality of the tool compared to implementing the same functionality ourselves. The advantage of using these libraries is that they have been in development for years, and many other projects use them, resulting in more discovered and corrected bugs. In comparison, our tool has been in development for three months and only tested by two people using unit tests. Moreover, realising the algorithms and measures mentioned in Chapter 2 would be infeasible in three months, particularly given the complexity of the used algorithms (e.g., t-SNE and MDS).

3.2 Research questions

The research questions addressed by the design-science research methodology are found below. As mentioned in the Introduction (Chapter 1), the first question is related to automated test prioritisation, and RQ2 relates to the visualisation of data used in the prioritisation process. The third research question combines the answers of RQ1 and RQ2 into a summary of the effects.

> **RQ1** How can we instrument CI pipelines to optimise test feedback cycles?
> **RQ1.1** What different types of information can we use to optimise tests in CI pipelines?
> **RQ1.2** What are the trade-off in using different types of information from test artefacts?
> **RQ1.3** How effective are the optimised test suites?
>
> **RQ2** How can we use test optimisation to support human decision making in test cycles?
> **RQ2.1** What type of information do stakeholders use to make testing decisions?
> **RQ2.2** To what extent can we capture and visualise this information to trigger insights from testing cycles?
>
> **RQ3** How does the integration of test optimisation in CI pipelines affect practitioner's feedback cycles?

When it comes to the first research question, its purpose is to find an idea or a way to use test optimisation in CI pipelines and thus improve test feedback cycles. This question consists of three subquestions (i.e., RQ1.1, RQ1.2 and RQ1.3) where each

of them corresponds to a different aspect of test optimisation and CI pipelines. For RQ1.1 we look at which types of information (e.g., execution history and test case names) can be used in test prioritisation with CI pipelines. With RQ1.2 we try to find if it is possible to use different test case data in test optimisation and how these affect the outcome from prioritisation. By answering RQ1.2, we can determine which test case data is best suited for test prioritisation in CI pipelines. Finally, with RQ1.3 the aim is to determine which test case data and distance measure produces the most effective test suite. With the answer from RQ1.3, we can provide better optimisation of the tool for use in CI pipelines.

By answering RQ2, we will learn about what data practitioners currently use, and what they would like to use when making decisions regarding testing. Moreover, the answer to RQ2 determines if we can obtain this data from the test prioritisation process and visualise it for a more in-depth understanding of test cycles. The sub-questions of RQ2 (i.e., RQ2.1 and RQ2.2) allows us to cover the different aspects of RQ2.

The last research question, RQ3, combines the answers of RQ1 and RQ2 to provide an understanding of the effects on feedback cycles when applying test prioritisation to CI pipelines. Our answer to this research question can present meaningful insights into test prioritisation as a part of the CI toolchain.

3.3 Evaluation and case study design

After the final iteration on our design science methodology, we evaluate our tool by conducting a case study, which involved first-degree data collection and an interview. These methods address RQ1 and RQ2 respectively, and with these solutions, we answer RQ3. The case study was performed together with Volvo Car Corporation where we conducted the first-degree methods.

Volvo Car Corporation is an excellent choice for conducting our study since it allowed us to involve a large company within the automotive domain instrumenting CI and continuous execution of tests. Additional factors in our decision regard their test rigs and execution times. More specifically, software testing is prohibitive due to the availability of the test rigs being low but is made worse by long execution times.

Our case study has two objectives: To find new insights regarding automated prioritisation and visualisation of data used in the prioritisation process. However, also to describe current practices regarding test and feedback cycles and changes that might come from the implementation of automated prioritisation and visualisation of its data. This case study involves an interview and collection of various information about tests (e.g., execution results) from the automotive company Volvo Car Corporation. A summary from the result of our planning is found in Table 3.1.

Objective	Exploratory and Descriptive
The context	Automated test case prioritisation in CI pipelines
The case	Volvo Car Corporation
Theory	Diversity-based optimisation and Software visualisation
Research question	RQ1, RQ2 and RQ3
Method	First degree data collection, and interviews + archival data
Selection strategy	Companies using CI pipelines
Unit of Analysis 1	Automated prioritisation APFD and Coverage
Unit of Analysis 2	Visualisation Satisfaction or insights triggered by the engineers we interview

Table 3.1: Planning of our case study using guidelines by Runeson and Höst [30]

In the subsections that follow, we will describe the first unit of analysis in Section 3.3.1. Next, we will present the second unit of analysis in Section 3.3.2 and then finish by addressing the case company in Section 3.3.3.

3.3.1 Unit of analysis 1: Automated prioritisation

For the first unit of analysis, our subject is a software project in a CI pipeline, which has at least 30 days of execution results available while at the same time not having a very high rate of failure. Preferably, our tool should detect more test cases than the average amount of test cases available in each project at our disposal. This filtering enable the identification of variations between treatments after prioritisation.

APFD is one of the measures used in the first unit of analysis, which determines the quality of prioritised test suites since APFD will tell us how early a test suite detects faults [31], [32]. When performing prioritisation, it results in the most critical test cases are executed first. In our case, the most diverse test cases execute first, resulting in higher chances of detecting different faults which is why APFD is essential for determining differences between our treatments. Another advantage of discovering faults early in the test suite, is that if the execution of all test cases is not possible, either by choice or some failure during execution, then we can rest assured that the most significant test cases were executed first. For a test suite T with n test cases, and if the software under test contains m faults, we can calculate the APFD value if we also let TF_i be the position of the first test case in T that exposes fault i:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{1}{2n}$$

As the meaning and name of APFD imply, it is based on faults; however, we do not have access to fault information for our subject (including candidate subjects). As

such, we base it on failures instead. Thus, we assume that each test case identifies a unique fault in the software that no other test case can find. Moreover, this assumption is supported by previous research who in a similar situation drew the same assumption, for example in a paper from de Oliveira Neto *et al.* [9].

Our second measure is fault coverage. It is calculated by dividing the number of faults that is identified by the test suite, with the total number of known faults in the SUT. More formally, let us consider m_T to be a number of faults in the SUT detected by a test suite T and m to be the total number of known faults in the SUT. With this, we calculate the fault coverage as:

$$Fault\ coverage = \frac{m_T}{m}$$

In our study, we calculated multiple fault coverage values for various sizes of test suites. We selected these subsets by choosing among the top priority test cases (i.e., ranked first), at different cut-off. In other words, we investigated effectiveness if we executed only the top-priority tests. In total, we gathered 19 subsets (excluding test suites containing the same test cases as the prioritised lists and empty test suites) from each prioritised list of test cases. The subsets were selected five percentage points apart, e.g., at 5%, 10% and 15% of the prioritised lists. For each cut-off value we repeated the selection 100 times to observe variation. This yields: $100 \times 19 = 1900$ executions per prioritisation technique.

We use feature coverage as the last measure in our study. It is described as the number of features out of all features in the SUT that is covered by the test suite in per cent. We calculate feature coverage by dividing the number of features covered by the test suite, with the total number of features in the SUT. In more formal terms, if we have a test suite T that covers f_T features out of the total number of features f in the SUT, then we can derive the feature coverage as:

$$Feature\ coverage = \frac{f_T}{f}$$

The calculation of feature coverage was performed in the same manner as with fault coverage, resulting in a feature coverage value being calculated for each of the 19 test suites derived from each prioritised list of test cases. As with fault coverage, we create 100 prioritised lists for each treatment.

The test repository at the case company used in this study presents the features that each test case evaluates in the name of their corresponding test suite. However, this might not apply to other test repositories at the company or within other organisations as well.

Abbreviation	Description
	Criteria
CODE	Criteria for the code of test cases
NAME	Criteria for the full name of test cases
SIMP	Criteria for test case names where the ECU name found in some test cases is removed
EXEC	Criteria for historical execution results of test cases
	Distance measure
JAC	Jaccard Index
LVS	Levenshtein
NCD	Normalised Compression Distance
XOR	The XOR based distance measure used for the execution history criteria
RDM	The random treatment used for comparison and analysis for the first unit of analysis

Table 3.2: Abbreviations used for describing aspects of the first unit of analysis, including results and analysis procedure.

Our study consists of multiple separate experimental designs. With the help of the abbreviations defined in Table 3.2, we are able to describe these designs. We have one factor (distance measure) with five levels (LVS, JAC, NCD, XOR and RDM). However, this factor is evaluated under different test case data (we refer to them as criteria). We did not treat the criteria as a factor because not all criteria are compatible with all distance measures. To keep the design simple, we run the comparison between criteria and distance measures separately. Table 3.3 presents an overview of our experimental designs.

No.	Criteria	Factor	Levels
1	CODE	Distance Measures	LVS, JAC, NCD and RDM
2	NAME	Distance Measures	LVS, JAC, NCD and RDM
3	SIMP	Distance Measures	LVS, JAC, NCD and RDM
4	EXEC	Distance Measures	XOR and RDM

Table 3.3: Overview of the experiment design for unit of analysis one.

We automated our data collection by implementing short scripts. Each script performs prioritisation of the investigated test cases for each treatment as seen in 3.3. Moreover, the resulting list from each prioritisation is in turn used by the script to calculate a number of observations for each trail.

After data collection, we performed a per treatment analysis of the possibility of assuming that it originates from a normal distribution using the *Shapiro-Wilk test* [33]. The importance of know if the samples originate from a normal distribution is vital in determining which test to use next. When it turned out that most samples allowed for rejection of Shapiro-Wilk's null hypothesis, the next phase of the analysis was to look at various non-parametric statistical tests. We settled for the *Kruskal-Wallis* [34] test to determine if all treatments in the same design come from the same

population. The choice of using Kruskal-Wallis for this part of the analysis is because it applies to unmatched groups which correspond to the data. Since Kruskal-Wallis only tells if there is a stochastic dominance present between the treatments in each design, we are required to perform two more steps of analysis before we can draw any conclusions. These two steps are performed pair-wise for each treatment in each design. The first step is to use *Bonferroni corrected Mann-Whitney* [35] tests to determine where any identified stochastic dominance is present. The second and final step is to calculate the actual effect size using the *Vargha-Delaney test* [36]. The result from the Vargha-Delaney test is the effect size of the samples.

3.3.2 Unit of analysis 2: Visualisation

For the second unit of analysis, we used convenience sampling, resulting in interviewing 6 subjects with different experiences of automated testing and CI pipelines. Even though the subjects were developers, they were suitable for participating in our interviews since they had enough knowledge and experience of testing. In table 3.4, the subjects' role and experience of software development are shown. We interviewed three CI infrastructure engineers, two Dev-ops engineers and one first analysis engineer[2]. Also, they had various experiences of software testing, where the least experienced developer, as participant A in the table, had seven months while the the most experienced, as participant D in the table, had as much as 13 years. In general, the subjects' experience is short since the case company only recently started with large scale software development, but also since there is currently a high demand for software developers in the area. Therefore, it was hard to get a hold of subjects with long experiences of software engineering.

Participant	Role	Experience
A	CI infrastructure engineer	7 months
B	Dev-ops engineer	2 years
C	CI infrastructure engineer	5 years
D	CI infrastructure engineer	13 years
E	Dev-ops Engineer	2 years
F	First analysis engineer	2 years

Table 3.4: Participants of the interview

Before we started to interview the participating subjects, we asked them for consent to participate in our study and if we could audio record them during the interview.

We prepared the interviews by generating questions for our initial instrument. An experienced third party assessed and gave feedback on our interview instrument that we later improved, which increased the construct validity of our study. We designed

[2]An engineer that analyses results from tests

the interview instrument so that the questions can be mapped to the corresponding research question. The complete interview instrument is presented in Appendix A. Before we asked any questions related to our research questions, we asked some introductory questions.

The introductory questions in the first part of the instrument interview is important to be aware of their role and experience. We also needed to ask these questions to know if the interviewees were really suited for our study. Moreover, we could also know if we interviewed subjects with too short experience and thus determine if they should be a part of our study to increase the reliability of our results. The introductory questions in the interview are also easy to answer and therefore could establish trust between us and the interviewees [37].

The second part of the interview instrument contains questions related to RQ2.2, but is also about evaluating the quality of our visualisations in the tool. When we created questions for this part, we used two papers to look at how the visualisation community constructed their interviews to evaluate their visualisations [38], [39], as seen in Appendix A. By doing this, we could make sure that we asked the right questions in order to evaluate our visualisations in the tool. When we asked questions related to this part of the instrument, we presented three different plots that the interviewees could interact with in order to give us useful answers from the interviews.

We included questions in the third part of the instrument interview to collect knowledge about the interviewees' experience of testing. For example, we were looking for approaches that they used to create new tests and if they used any testing process. Additionally, the questions raised discussions about the type of information they were using to make testing decisions, which is necessary for our RQ2.1. We also made it clear for the interviewees that the testing decision could be any testing decision, such as testing strategies to what tests they wanted to run, create or delete.

When it comes to the last part of the instrument interview, it consists of questions related to RQ2.2, which raised discussions about the interviewees' experience of test optimisation. The questions are formulated in such a way that it is clear to understand that we are looking for both manual and automated test optimisation. Moreover, they helped us to dig into their opinions of how their way of working would be affected by using test optimisation and if they saw any drawbacks of using test optimisation in their daily work.

When we finished with all our interviews, we performed thematic analysis of the qualitative data from all our six subjects. We followed guidelines from various papers [40], [41] where they define the following six steps for performing thematic analysis, including our work at each step:

- *Step 1 - Get familiar with your data.*

  Since we audio recorded the interviewees with their consent, we started to transcribe each recording by listening to each audio file in small sections multiple

times. Although this required much time, we could make sure that we did not miss any words or take any incorrect words. Besides that, we also took notes of some important details that came up during the interviews. By details, we mean something that was hard to catch up from the audio recordings, for example when the interviewees pointed at the different plots that were used in the interviews.

- *Step 2 - Generate initial codes.*

  The results of transcribing the audio recordings were used and transferred to NVivo[3]. We used this program to be able to write down codes for all our transcripts that covered all details of the interviews. We agreed on our codes and completed our code assignments by thoroughly discussing it. This resulted in higher conclusion validity of our coding and thus the results from our thematic analysis. We also removed some codes since they did not contribute to answering our research questions.

- *Step 3 - Identify themes.*

  Nvivo were also helpful for creating themes. At the beginning, we created draft themes by grouping all codes that belonged to each other or at least had something in common. Later, we discussed whether these themes made sense. By doing this, we could quickly find useful information for our research questions.

- *Step 4 - Evaluate identified themes.*

  Our themes were evaluated by an experienced third party so that we could make sure that we performed the thematic analysis correctly, which increased the reliability and conclusion validity of our study.

- *Step 5 - Define and name the themes.*

  The name of the themes were finalised based on their context and feedback received. Also, we moved around codes to other themes to make sure that every code belonged to the right theme. Afterwards, a thematic map was created to present the result of our thematic analysis.

- *Step 6 - Create the report.*

  With the help of all our codes and themes, we could select relevant extracts that we believed best represented the opinions of our interviewees. These extracts were later used to relate to our research questions in the report.

[3]NVivo - https://www.qsrinternational.com/nvivo/home

3.3.3 Case Company

The study was performed at a group of approximately 20 people working with test automation at Volvo Car Corporation (VCC). The majority of them are testers, but some developers are present in the group. The group has a number of CI pipelines for each software project and they use Jenkins to integrate and test each developer's code into a share repository. Each CI pipeline corresponds to a type of software testing (e.g., acceptance testing and integration testing) so that each project can have multiple types of software testing. All test cases in the test repository are executed 4 times per day, which are controlled and triggered by a build system. The test cases are executed on test rigs consisting of affected ECUs and various peripheral devices, whose hardware needs to be configured to suit the particular project. Robot Framework[4] is used to create the test cases, which are written in natural language by humans. There are approximately 523 test cases on average per project and each execution of a full test suite lasts for about 116 minutes on average. For the project analysed in this thesis, our tool have identified 183 integration level tests and their execution time is approximately 93 minutes, which is close to the average test run.

[4]Robot Framework - https://robotframework.org

4

Tool

The purpose of this chapter is to explain the toolkit developed and used in this study. The chapter contains three subsections: the description of the practices used to create the tool and how it operates, another one detailing the prioritisation algorithm used by our tool and finally a section detailing how the tool uses dimensionality reduction and other techniques to visualise test data.

4.1 Operation

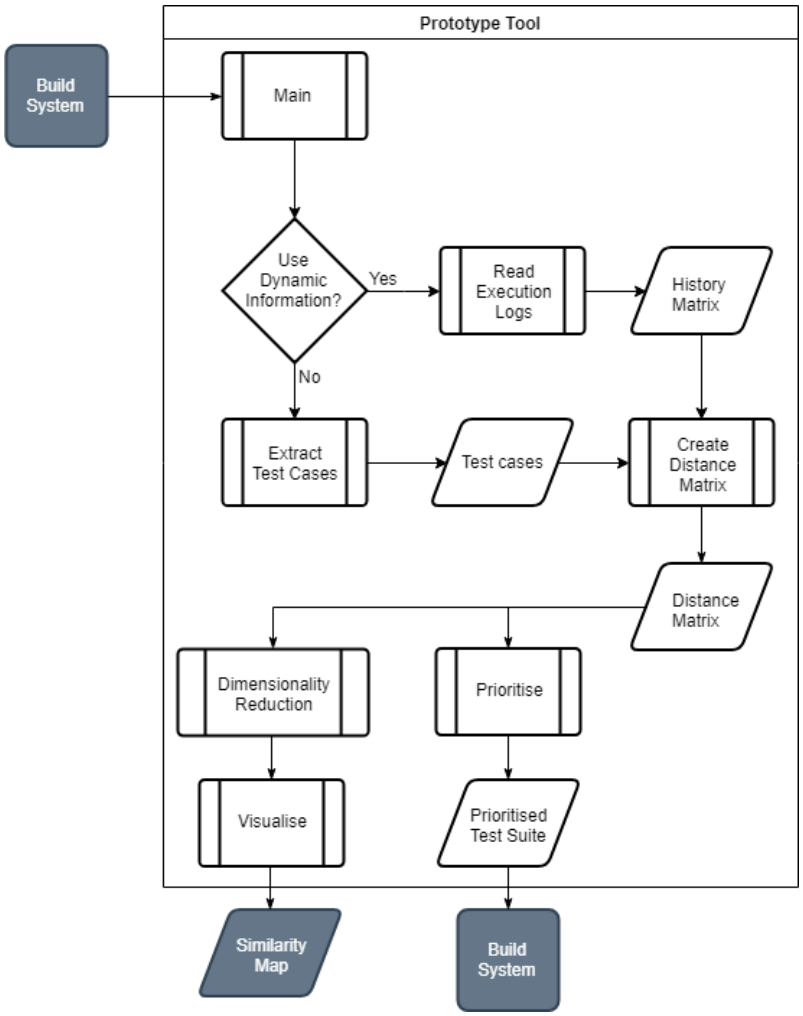


Figure 4.1: Overview of our tool workflow when test execution has been triggered in the build system

Figure 4.1 shows the flow of the tool when a build system triggers execution of test prioritisation in a CI pipeline (e.g., a new build starts in Jenkins). The darker entities in the figure represent external systems and output from the tool. When the build system receives a commit, it triggers the prototype tool's Main process which determines what algorithm, criteria and encoding strategy to use as the basis for prioritisation.

If the Main process decides to use dynamic information, it starts the Read Execution Logs process. If the Main process should use dynamic information, it starts the Read Execution Logs process. This process starts by downloading any missing logs before it locally reads all related logs and creates a history matrix that it outputs to the Create Distance Matrix process. A history matrix represents the execution results of

test cases over time, which will be further detailed when discussing the architecture of the tool later in this section. The distance matrix serves as a basis for both the Prioritise and the Dimensionality Reduction processes. The prioritisation process performs test case prioritisation, and the result of this is a prioritised test suite which the tool sends back to the build system. In the meantime, the Dimensionality Reduction process will apply either MDS or t-SNE, at the users' discretion, on the distance matrix to perform dimensionality reduction. As the output of this process, each test case is mapped to a coordinate in a two-dimensional plot, i.e., the Similarity Map, included in the visualisation toolkit.

However, if the Main process instead decides to use static information for the distance matrix, it will trigger the Extract Test Cases process. This process reads and extracts test cases from the hard drive which it uses to start the Create Distance Matrix process. This process will then calculate the pair-wise diversity for all test cases and place the results in a distance matrix. The resulting distance matrix is then used in the same way described in the previous paragraph for a distance matrix created using a history matrix.

4.1.1 Prioritisation algorithm

The purpose of the prioritisation algorithm is to identify the most diverse test cases in a test repository and sort them accordingly. For the prioritisation algorithm to be able to do this, it requires a distance matrix. Our prioritisation algorithm starts by identifying the highest value in the matrix, which represents the most dissimilar pair of test cases in the matrix. As we discussed in Section 2.2.1, the distance matrix is symmetric, hence it does not matter if the algorithm decides to examine distance values from the upper or lower triangle of the matrix when identifying the highest value since the algorithm will still work with the upper triangle of the matrix in the next steps of the prioritisation process. As a reminder, the goal is to find the most diverse *pairs* of tests. Note that different pair of test cases can be equally diverse. That implies that the matrix contains pairs with the same distance values. As a tie breaker, our algorithm will choose a pair at random to be examined in the next part of the prioritisation process.

When the highest diversity value (i.e., most diverse pair of test cases) is found in the matrix, the goal is to choose one from the corresponding pair. For each one, we sum all it's corresponding diversity values (i.e., the entire column/row in the matrix). The test case with the highest sum is identified as the most diverse of the two with regards to the other test cases in the matrix. This sum is the summary of all pair-wise distances in the matrix that includes the test case. However, we need to observe that the algorithm only includes the cells above the main diagonal of the distance matrix when calculating the sum. If the algorithm included the duplicated values in the sum, it would result in additional and unnecessary calculations in an algorithm whose execution time we want to minimise. As such, there is a potential

that these additional calculations could become noticeable for huge test repositories.

When the algorithm has identified the most diverse test case of the pair, it appends this test case to the prioritised test suite and removes its column and row from the distance matrix. However, an exception to this occurs when the two test cases have equal sums. In this case, the algorithm will randomly choose one of the two test cases that it will append to the prioritised test suite and remove from the distance matrix.

The algorithm repeats the process described above until the highest value in the distance matrix is zero. At which point the algorithm randomly orders the remaining test cases and appends these to the prioritised test suite. The reasoning behind this is that if the highest value in the distance matrix is zero, it means that the remaining test cases are identical, and thus, their ordering does not matter.

We will illustrate the described algorithm above with an example. The example contains five test cases whose ID and name are found in the table below (Table 4.1)

ID	Name
A	*aaaa*
B	*aaaa*
C	*bbbb*
D	*aaaa*
E	*bbba*

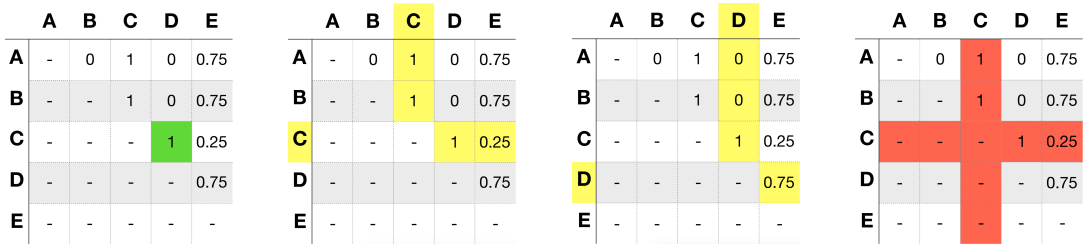
Table 4.1: Five example test cases with name and ID.

	A	B	C	D	E
A	-	0	1	0	0.75
B	-	-	1	0	0.75
C	-	-	-	1	0.25
D	-	-	-	-	0.75
E	-	-	-	-	-

Figure 4.2: The distance matrix before prioritising the test cases A, B, C, D and E. Values located above the main diagonal, are also present below it in practice, due to symmetry. However, we replaced them with hyphens in the example to reduce unnecessary clutter.

The matrix in Figure 4.2 contains three pairs of test cases, (A, C); (B, C); (C, D), with the same diversity as the highest value, 1, of the matrix. At which point, the prioritisation algorithm will randomly select one of the three pairs to bring to the next part of the process. In this example, the process selects (C, D), as seen in Figure 4.3a below. The total diversity of test case C is the summary of all values highlighted with yellow in Figure 4.3b. We calculate the sum associated with test case D in the same way, as seen in Figure 4.3c. The total diversity of test case C is 3.25 and 1.75 for D. Therefore test case C will be placed first in the prioritised

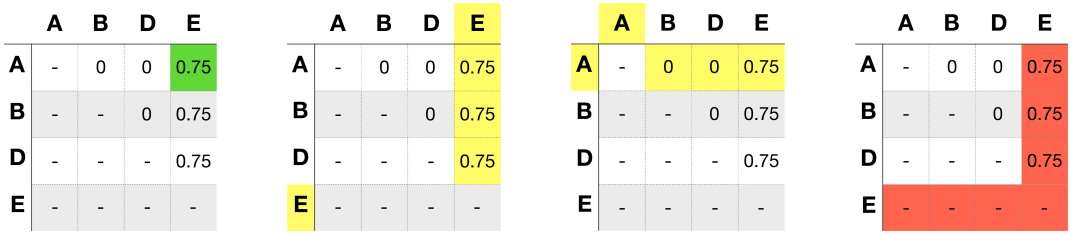
test suite due to its higher sum. Moreover, the row and column associated with test case C are removed from the distance matrix as shown in Figure 4.3d.



(a) Select C and D **(b)** Summarise C **(c)** Summarise D **(d)** Remove C

Figure 4.3: The steps required to add the first test case to the prioritised test suite

Now we can see in Figure 4.4a below that the distance matrix is smaller and contains only four test cases with the highest value being equal to 0.75. Since the matrix contains three pairs of test cases with this value, the algorithm will randomly select one of them, which in this case is test case E and A as seen in Figure 4.4a. After calculating the sums as presented in Figure 4.4b for test case E and Figure 4.4c for A, we find that the total diversity of test case E is 2.25 and for A it is 0.75. At this point, the algorithm will append test case E to the prioritised test suite before removing its row and column from the distance matrix, as seen in Figure 4.4d.



(a) Select A and E **(b)** Summarise E **(c)** Summarise A **(d)** Remove E

Figure 4.4: The steps required to append the second test case to our prioritised test suite

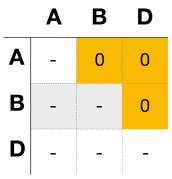


Figure 4.5: Test cases remaining after adding two test cases to the prioritised suite, and all have equal diversity

At this point, the distance matrix as seen in Figure 4.5 above, contains only diversity values equal to zero. The result of this is that the algorithm randomises the order in which it appends the three remaining test cases to the prioritised test suite. In this example, it first appends test case B, before appending test case A and finally

test case D. With this randomised order, the resulting prioritised list can be found below (Table 4.2.

Priority	ID	Name
1	C	*bbbb*
2	E	*bbba*
3	B	*aaaa*
4	A	*aaaa*
5	D	*aaaa*

Table 4.2: The resulting prioritised list from the prioritisation example.

4.1.2 Visualisation

Our visualisation tool contains two types of plots: a heat-map styled plot based on historical data (called *History plot*) and a similarity map which places a distance matrix in a dimensionality reduction technique. To plot these types of plots, we used a library called *Plotly*[1] in our tool.

We use a heat-map to visualise history matrices based on their execution results. The x-axis of the heat-map represents execution runs, while the y-axis represents unique test cases. The execution runs are sorted by the time-stamp (e.g., date) the test cases were executed in ascending order, similar to the case with the history matrix. The heat-map uses different colours for the different execution results so that red corresponds to a failure, green to a pass and white for test cases that did not execute during a run. Whenever there are enough number of test cases and execution runs, the heat-map can help with providing an overview of execution results for different test cases over time. Several patterns in the plot can also be identified since it shows relationships between different test cases, but also between different execution runs.

There is an example of a heat-map styled plot in Figure 4.6 that is based on the execution results from the selected project. First we can see that there are some white vertical lines in the plot (e.g., between execution runs 29 and 37), which is due to test rig failure resulting in no test cases executed. Other than this, the test case number 10 is one of the most successful test cases in the plot, while the test case number 150 is one of the least successful ones. There are also some interesting phenomena appearing around test case number 90. Test case 90 fails every time it is executed, and so does e.g., test case 80 for a while as well. That is, until execution run 41 they are identical with respect to dynamic information. After the 41st execution, test case 80 instead passes almost every time it executes. As a result, test case 80 becomes very distant to test case 90 while growing steadily more identical to test cases that rarely fail (e.g., test case 89). What makes this even more interesting is that this is true for a lot of test cases.

[1]Python Graphing library, Plotly - https://plot.ly/python/

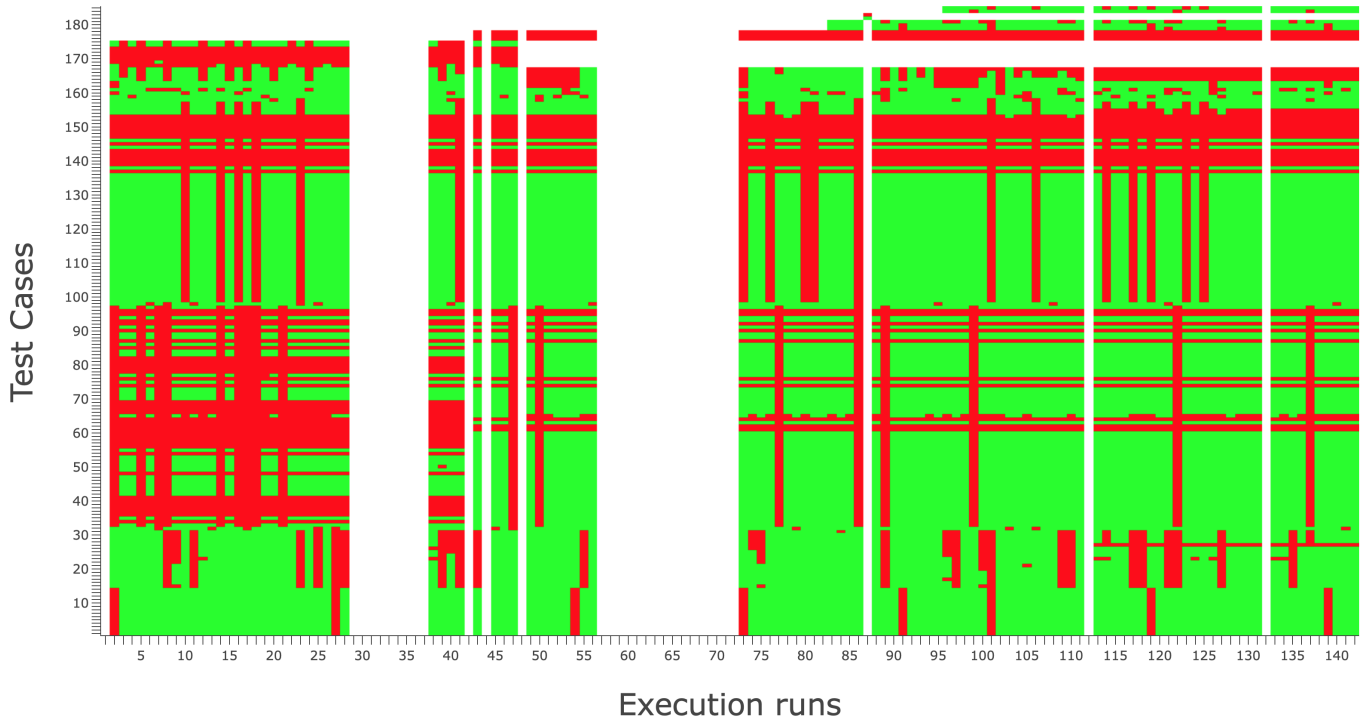


Figure 4.6: A heat-map visualisation of execution history for the test cases in this project

In our prototype tool, we use a scatter plot to visualise distance matrices as similarity maps. The exact position of a test case in the plot does not matter, instead only its distance and relationship with other test cases do. Our tool can plot distance matrices by turning them into two-dimensional maps using dimensionality reduction [9]. The two-dimensional map together with Plotly creates a similarity map. Our implementation of dimensionality reduction in our tool supports both t-SNE and MDS.

Figure 4.7 presents a similarity map of a data set mined from the test cases created by the Mozilla Browser Front-end QA team[2] team. The similarity map is based on a distance matrix calculated by using Levenshtein and is visualised by using t-SNE to reduce the dimensions of the distance matrix to two dimensions for easy visualisation. The t-SNE algorithm ran with a perplexity of 60 and number of iterations of 2000, and the distance matrix comes from static information as test case names. It is possible to identify some larger clusters in the plot, e.g., the largest cluster at the bottom right of the plot, and some smaller clusters, e.g., the small cluster above the largest cluster to the right. The plot shows clearly the different clusters and makes it easy to identify test cases that are similar to each other based on static information as test case names.

[2]https://www-archive.mozilla.org/quality/browser/front-end/testcases/ Mozilla Browser Front-end test cases

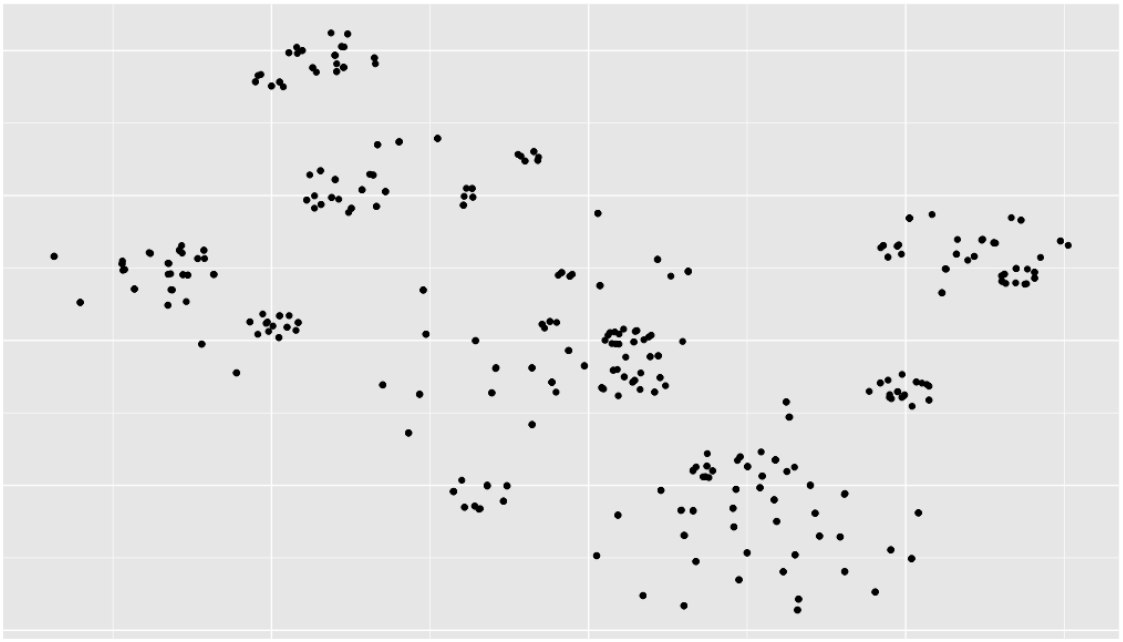


Figure 4.7: Similarity map of test case names using t-SNE for dimensionality reduction. The clustering of tests in this case, represent tests covering requirements with similar names.

Figure 4.8 presents the same distance matrix as Figure 4.7, but instead of using t-SNE, this plot uses MDS to reduce the dimensions of the distance matrix. The plot shows that the more we go to the left of it, the more are the test cases abandoned from each other and the more we go to the right of it, the more are the test cases clustered together. We can see in the plot that the MDS output is not as clustered as it is with t-SNE, which makes it harder to identify multiple test cases that are similar to each other.

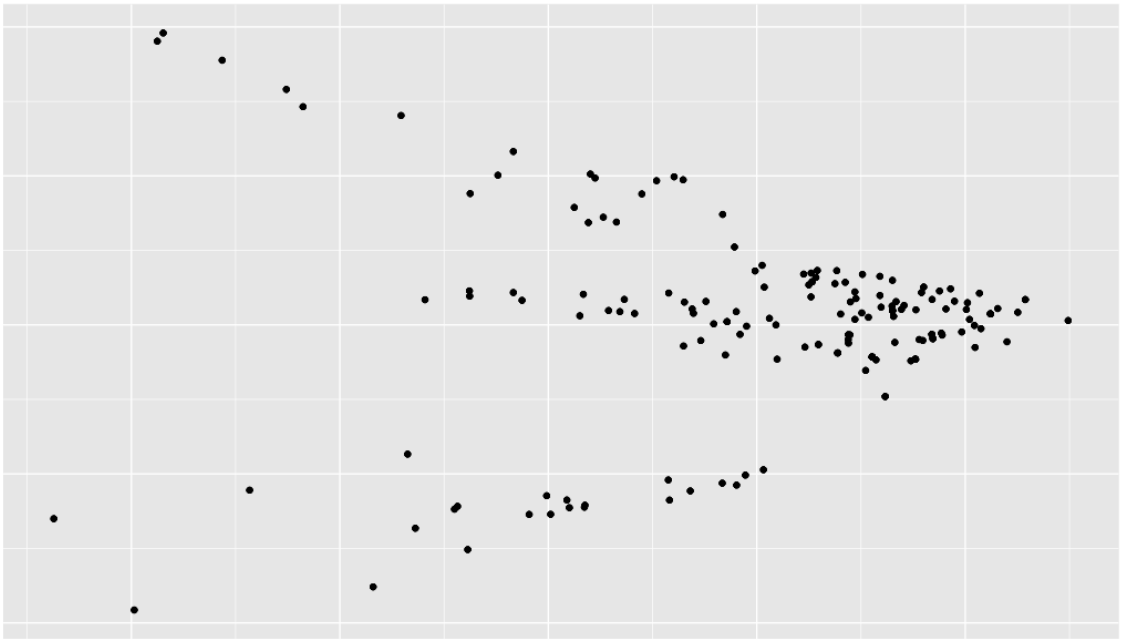


Figure 4.8: Similarity map of test case names using MDS for dimensionality reduction. The clustering of tests in this case, represent tests covering requirements with similar names.

4.2 Architecture

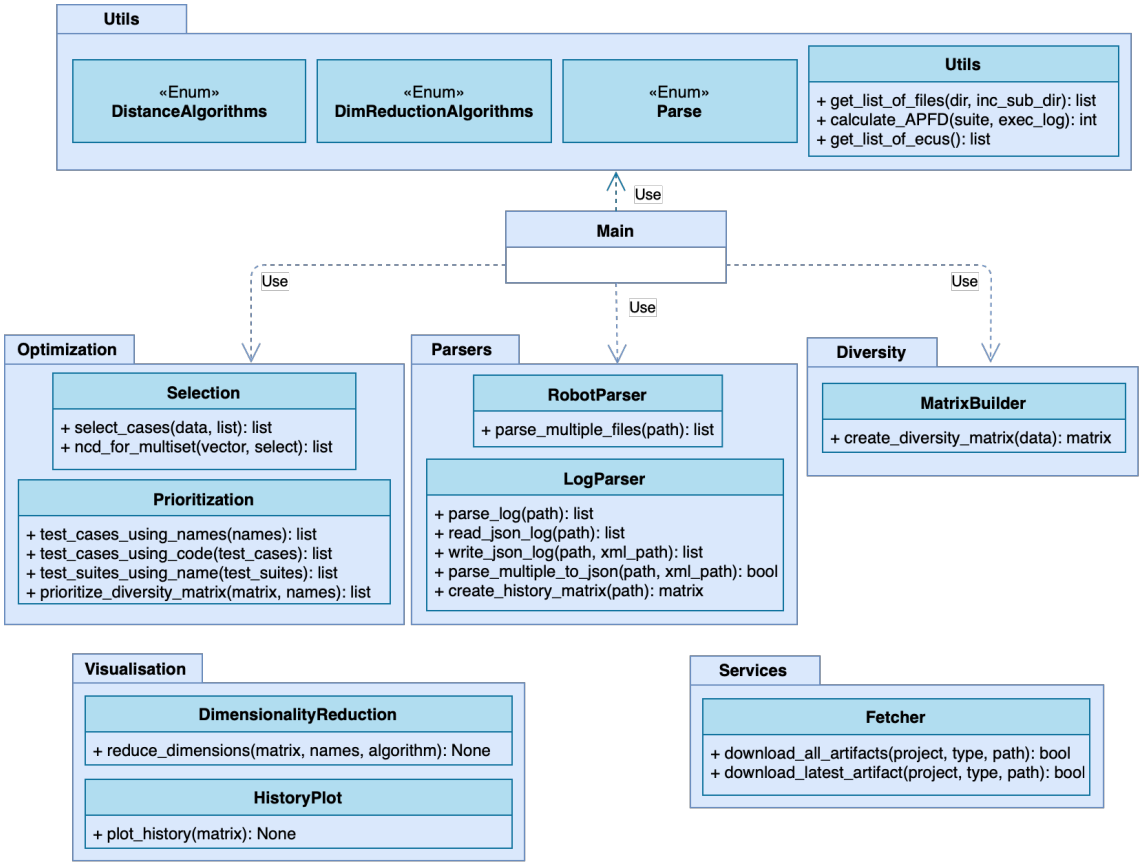


Figure 4.9: The architecture of the tool

The architecture of the tool, as seen in Figure 4.9, was created with the intention of a modular system that is easily extended to work with other systems than those currently present at the company. It results in the Main class calling various packages and modules in the tool, whose purpose is to perform the required measurements and data gathering for this study.

Starting from the top of Figure 4.9, the Utils package contains functionality that almost all of the packages and modules in the tool requires. Because of the extensive use of this package, we did not include them in the architecture presented in Figure 4.9 since it would result in a much more complex diagram. The Utils package contains enums for the available algorithms used in distance calculation and dimensionality reduction and an enum for specifying what information the tool should parse from Robot Framework test cases (e.g., test case name, test steps and test suite name). Moreover, the Utils package contains some methods for listing files and calculating the APFD value of a prioritised test suite.

In the Optimisation package, there are two classes, Selection and Prioritisation.

These classes enable the use of test case selection and prioritisation based on diversity matrices.

The Parsers package contains parsers for Robot Framework log-files and test suite files. The LogParser-class does not only parse log-files but also convert these to JSON files to minimise storage usage since the default log files are huge. Furthermore, the LogParser supports creating a so-called history matrix that combines the information from selected logs into a matrix. Each row in the matrix represents a unique test case, while each column represents one execution run. The result for each test case and run, i.e. a cell in the matrix, is represented by an integer. For a test case i and an execution run jx, its value $a_{i,j}$ in a history matrix is defined as:

$$a_{i,j} = \begin{cases} -1, \text{ if the test case did not execute} \\ 0, \text{ if the test case failed} \\ 1, \text{ if the test case passed} \end{cases}$$

Each new execution run results in appending a column to the matrix, which results in the columns of the matrix being sorted in ascending order with regards to time. Even though the columns are time, they represents that the tests were executed in the same version of the SUT. This lets us draw a important conclusion regarding dynamic information, which is that is two test cases always executes together on the same SUT and presents the same result, then they should always reveal the same fault. The resulting history matrix is the basis for creating diversity matrices based on dynamic information and for plotting execution history.

The Diversity package contains the MatrixBuilder class which creates diversity matrices using either static or dynamic information, including calculating the distances using a selection of pair-wise diversity measures. The function creating distance matrices uses a pool of processes to increase the speed of the calculations since matrices measuring as little as 1000×1000 requires about 500 000 distance calculations for only 1000 test cases.

The Visualisation package contains two classes, DimensionalityReduction and HistoryPlot, which can visualise diversity and history matrices respectively. The former uses either t-SNE or MDS to turn a distance matrix into a two-dimensional plot, while the latter uses a history matrix to create a heat-map plot based on execution results.

Lastly, the Services package contains the class Fetcher which is used to download execution logs from a system at the company that we are then able to append to our JSON-logs.

5

Evaluation result

Since the collection and analysis of unit of analysis 1 and 2 has been completed, the results of it will be presented in this chapter. The first unit of analysis concerns the automated prioritisation and the statistical analysis associated with the data gathering from repeated prioritisation of the sampled project (see Section 3.3.1). Meanwhile, the second unit of analysis regards the interview and the visualisations that our tool is able to provide (see Section 3.3.2). The chapter is divided into two sections; Automated prioritisation and Visualisation. When it comes to the first section (Section 5.1), four different types of results will be presented; APFD, fault coverage, feature coverage and execution time. Meanwhile, for the second section (Section 5.2) we present the outcome from unit of analysis 2 including quotes and reasoning gained during the interview sessions.

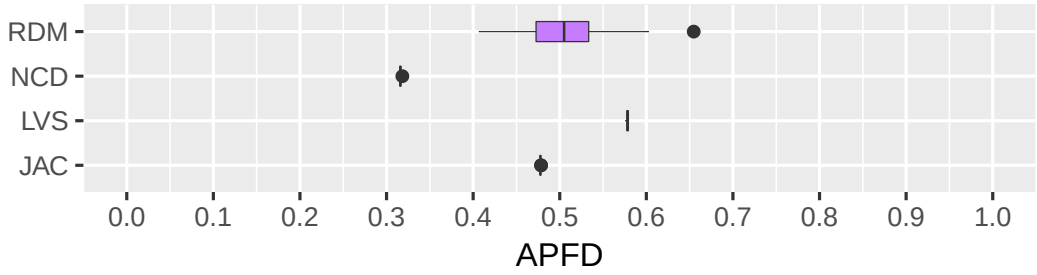
5.1 Automated prioritisation

This section presents the results regarding the first unit of analysis. We begin with Section 5.1.1 which presents the APFD, fault coverage and feature coverage plots. In Section 5.1.2, we present the statistical analysis on the coverage data. Finally, we present data on the execution times for the various treatments within this study, in Section 5.1.3. This data enables further discussions on the efficiency of the treatments.

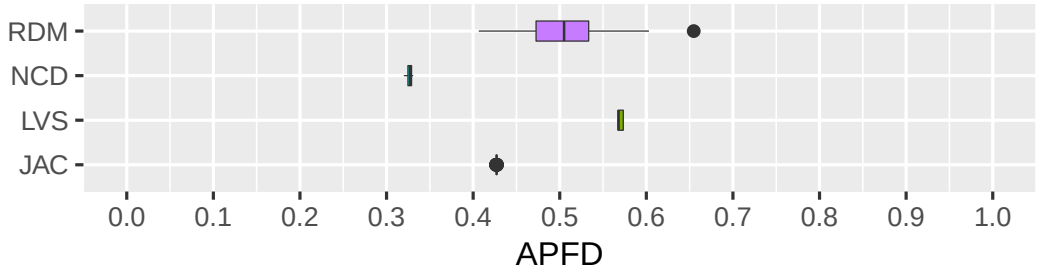
5.1.1 Visual analysis

In Figure 5.1, four different boxplots are shown, where each boxplot corresponds to a separate experimental design with a specific criteria as presented in Table 3.2. The x-axis of each boxplot represents APFD in fractions while the y-axis represents the different treatments used for each experimental design. Note that higher APFD values are better and mean that the prioritised test suite reveals, on average, faults earlier than the other test suites. All of these boxplots include four treatments (i.e., `RDM`, `NCD`, `LVS` and `JAC`), except the boxplot in Figure 5.1d with two treatments (i.e., `RDM`, `XOR`). `RDM` in all of these boxplots share the same APFD data.

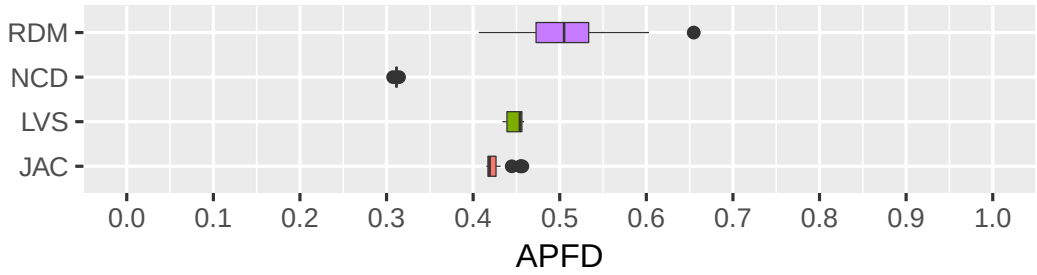
When it comes to the first boxplot regarding NAME, as seen in Figure 5.1a, all the APFD values from JAC and LVS are within the range of RDM. We can also see that RDM has a high variability while it for the other treatments is almost nonexistent. The result of this is that RDM can (but with low probability) produce test suites with higher APFD, than LVS, but also, as is more probable, test suites that have lower APFD than JAC. However, this only applies to our data set. When it comes to NCD, its APFD value is about 45 per cent less than LVS. The median for RDM is approximately 51%, for NCD it is about 32%, for LVS it is ca. 58%, and for JAC it is approximately 48%.



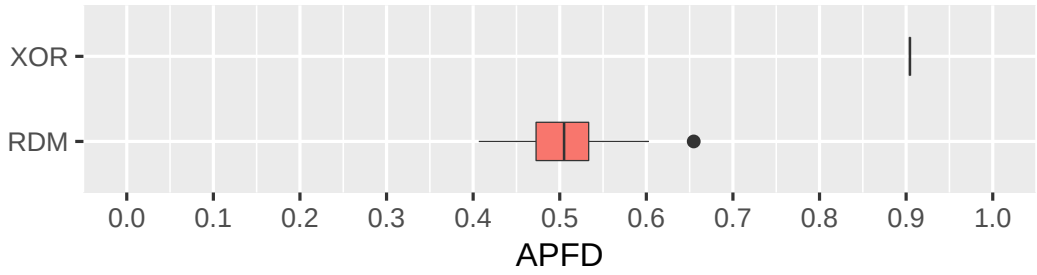
(a) The boxplot above is for the `NAME` criteria.



(b) The boxplot above is for the `SIMP` criteria.



(c) The boxplot above is for the `CODE` criteria.



(d) The boxplot above is for the `EXEC` criteria.

Figure 5.1: APFD results in the form of boxplots for all of our experimental designs and treatments.

The second boxplot in Figure 5.1b presents the APFD values of the same treatments shown in the previous boxplot, but the criteria is now `SIMP` instead. When comparing this boxplot with the first boxplot, they are very similar to each other, except that there are some slight differences in the treatments. One difference is that the APFD values from `JAC` are now around 42.5% instead, which means that they are lower

than the data points of the same treatment presented in NAME. They are also more far away from RDM's median. Moreover, the data sets from LVS and NCD have a slightly higher variability and show a sign of skewness.

The third boxplot, which is shown in Figure 5.1c, has still the same treatments as in the previous boxplots, although the criteria is CODE. The differences between the APFD values of the treatments in this third boxplot are much smaller compared to the previous boxplots. The APFD for all of these treatments are below the median of RDM. Furthermore, the data sets from LVS and JAC show a sign of skewness and their variability has marginally increased compared to the previous boxplots.

Figure 5.1d presents the fourth boxplot that has two treatments, namely XOR and RDM, where the criteria is EXEC instead. When it comes to the APFD values from XOR, they are considerably higher than the data set from RDM, which are around 90%. However, the variability of those values is very small.

Moving on to the second measure, namely fault coverage, four fault coverage plots are displayed in Figure 5.2. They represent how much per cent of faults are covered for every 5 percentage point selected from the prioritised list. The subsets selected from the prioritised list consist of test cases that are the most important ones. Each fault coverage plot shows a specific criteria and its associated treatments. The shadowed areas around the lines in Figure 5.2 shows the 95% confidence interval of the measured coverage. Since fault coverage is closely related to APFD, the APFD values of the treatments can be calculated by assessing the area under the curve in the fault coverage plots.

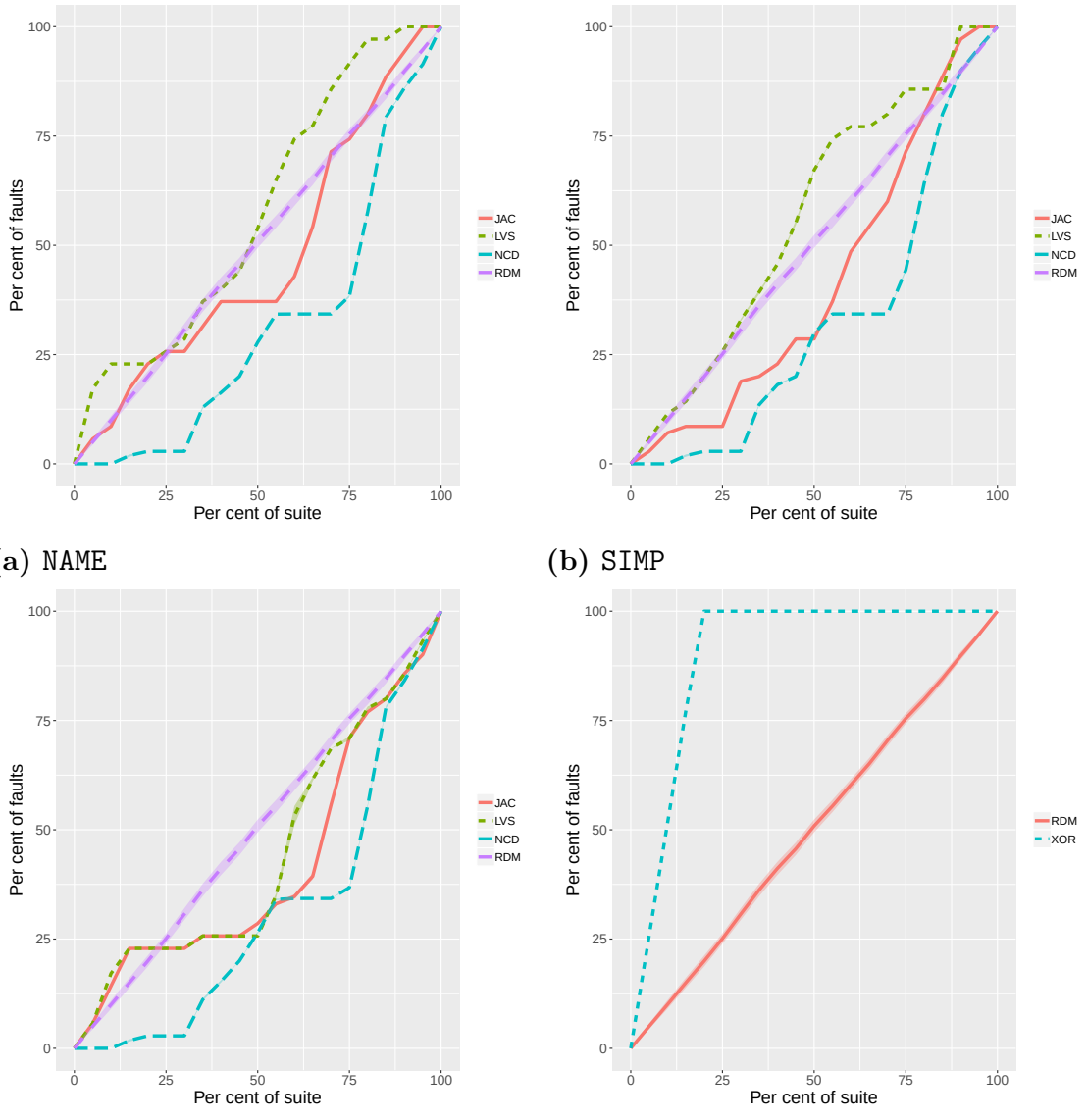
In the first fault coverage plot, as presented in Figure 5.2a, the percent of faults detected by suites created using NCD is always lower than that of the other treatments. The largest difference between NCD and the top performing treatment (LVS) is approximately 58 percentage points at 70 per cent. Notice that for the majority of the selected suites, LVS performs better than RDM and when it does not, it performs equally well as RDM (between about 25 and 50 per cent suite selection). On the contrary, for the majority of the selected suites, JAC performs equally well as RDM, and at other times, it has lower fault coverage. The most significant distance between JAC and RDM is approximately 18 percentage points and occurs when around 50 per cent of the suite is selected.

The second plot (Figure 5.2b) presents the SIMP criteria. For the majority of the time, LVS is the treatment with the highest fault coverage, where it between 0 and 25 per cent performs equally well as RDM. Given the small difference between the underlying data of SIMP and NAME, the difference in fault coverage, and how it changes as more of the suite is selected, is much greater.

When it comes to the CODE criteria, as found in Figure 5.2c, there are a number of interesting things that differ from the previously discussed graphs. At approximately 55% suite selection, NCD, LVS and JAC detect about the same amount of faults. This is also the one of the few times where NCD detects marginally more faults

than JAC, the other time being at 90% suite selection. Moreover, CODE is the only criteria where LVS is almost always lower than RDM. LVS and JAC discovers only more faults around 20 per cent. Additionally, JAC and LVS achieve almost identical fault coverage throughout every per cent of suite selection, except between 55 and 75% suite selection where LVS makes a higher increase in fault detection than JAC.

For the EXEC criteria, as seen in Table 5.2d, XOR has an linear slope that drastically increases to 100 per cent after only 20 per cent suite selection. At this point it detects 5 times more faults than RDM.



(a) NAME **(b)** SIMP

(c) CODE **(d)** EXEC

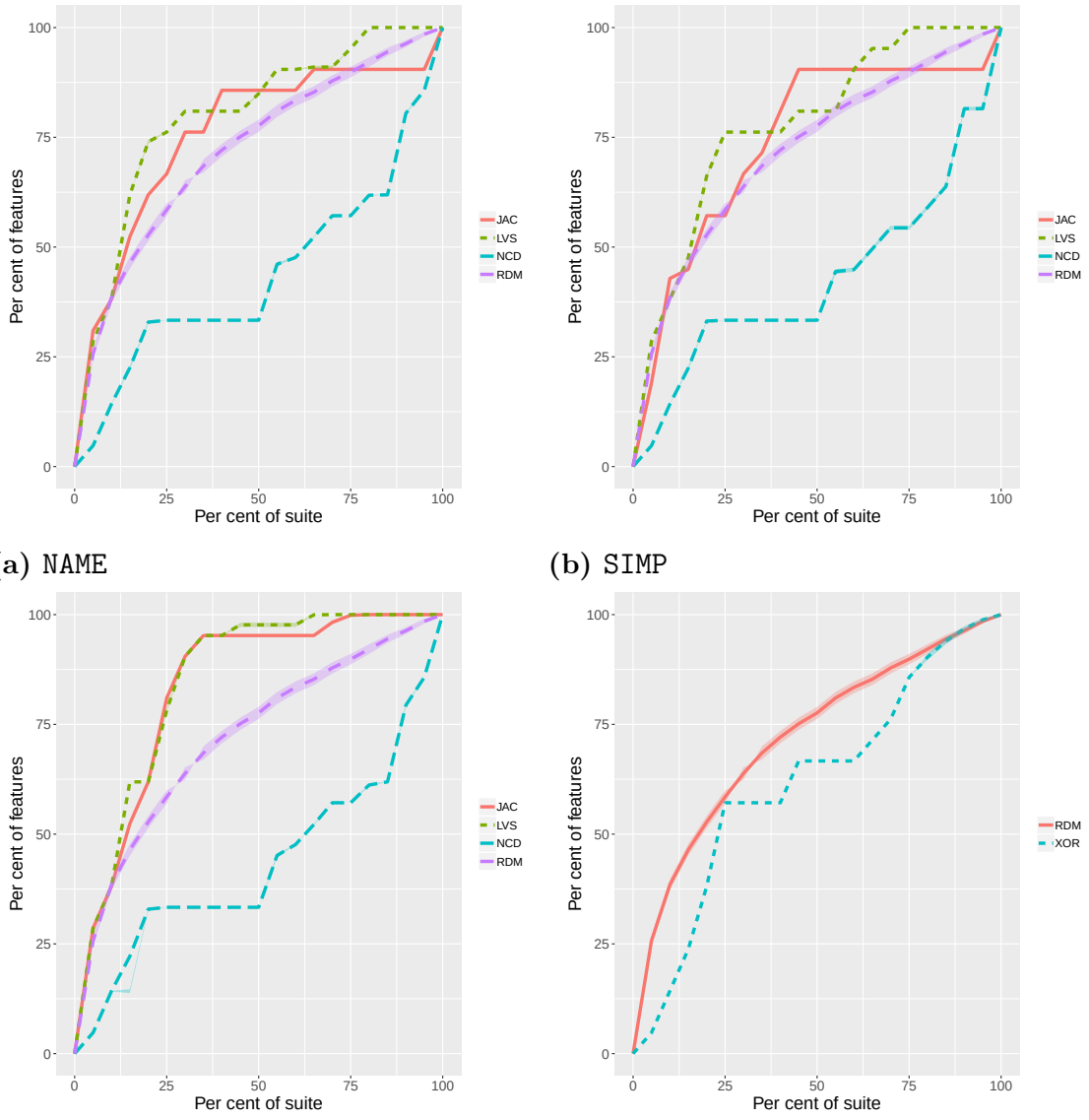
Figure 5.2: Fault coverage plot showing how our different criteria and treatments perform. The plot presents x per cent most highly prioritised test cases to be selected and how many faults are discovered by this selection.

When it comes to feature coverage, Figure 5.3 shows four feature coverage plots

representing how many features out of all features are covered by the subsets of the prioritised lists. The shadowed areas around the lines in the figure shows the 95% confidence interval of the measured coverage.

Figure 5.3a, presents feature coverage for the `NAME` criteria. Note that `JAC` and `LVS` very early reach high feature coverage, but after about 35 per cent suite selection, this increase begins to slow down. `LVS` reaches 100% feature coverage much earlier than the other treatments. With 50% suite selection, `NCD` has only about 30 per cent feature coverage and the distance between the other measures is at its peak. Here, `LVS` and `JAC` have about 55 per cent more coverage than `NCD`.

For the `SIMP` criteria (as seen in Figure 5.3b), all treatments are very similar to how they performed with the `NAME` criteria. However, the difference between `NCD` and `JAC` is more significant. With 50 per cent suite selection where the difference is at its highest, `NCD` achieves 30% coverage, while `JAC` has three times as high coverage.



(a) NAME

(b) SIMP

(c) CODE

(d) EXEC

Figure 5.3: Feature coverage plot showing how our different criteria and treatments performs. It selects the x per cent most highly prioritised test cases and sees how many of the features they target.

In the earlier feature coverage plots, JAC and LVS have presented very different feature coverage. However, for the CODE criteria (Figure 5.3c), they present very similar coverage and only differ around 20 per cent between 40 and 75 per cent of suite selection. Moreover, JAC reaches 100% feature coverage before LVS does, which has not happened for previous criteria on feature coverage. Meanwhile, NCD has not changed significantly compared to the NAME and SIMP criteria.

In contrast to what we saw in 5.2d, XOR never achieves higher feature coverage than RDM for the EXEC criteria in Figure 5.3d.

The visual analysis can be summarised with the following findings:

- For APFD, `LVS` has higher values than `JAC` which in turn has higher APFD than `NCD`. However, the only difference between them is that `NCD` is the only distance measure that has less APFD than `RDM`. The APFD values from `XOR` are considerably higher than `RDM`.
- When it comes to fault coverage, `NCD` has less fault coverage than the other distance measures. For `CODE` criteria, `NCD`, `LVS` and `JAC` have the same amount of fault coverage at 55% suite selection. With 20% of the suite selected, `XOR` achieves 100% fault coverage in the `EXEC` criteria.
- Regarding feature coverage, `JAC` and `LVS` have approximately the same coverage and achieve higher coverage than `RDM`. On the other hand, `NCD` detects considerably less feature coverage than `RDM`. Finally, `XOR` has less feature coverage than `RDM`.

5.1.2 Statistical analysis

As mentioned in Section 3.3.1, the statistical analysis begins with performing the Shapiro-Willks normality test on our samples. The null hypothesis of the Shapiro-Willks normality test is that the sample comes from a normally distributed population. With the data detailed in Table 5.1 we can reject the null hypothesis in favour of the alternative. In other words, none of our samples comes from a normal distribution. The effect of this normality test is that we will use non-parametric tests when analysing the effects.

	Feature coverage							
	CODE		Name		SIMP		EXEC	
	p-value	H_0	**p-value**	H_0	**p-value**	H_0	**p-value**	H_0
JAC	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
LVS	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
NCD	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
RDM	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
XOR	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
	Fault coverage							
	CODE		Name		SIMP		EXEC	
	p-value	H_0	**p-value**	H_0	**p-value**	H_0	**p-value**	H_0
JAC	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
LVS	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
NCD	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
RDM	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected
XOR	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected	2.2E-16	Rejected

Table 5.1: Presents the results from performing the Shapiro-Willks normality test on our data. The result means that it is possibly to reject the null hypothesis for all coverage samples with $\alpha = 0.05$. In other words, none of the samples follow a normal distribution.

Using Kruskal-Wallis we were able to identify that for each experimental design the treatments within are not from identical populations and applies to both feature and fault coverage data (p value less than $2.2E-16$ for each experimental design and measurement). This outcome means that we need to further investigate the samples to identify where the differences lies, and how large they are.

The final step of the statistical analysis is a pairwise comparison of effect sizes for all treatments and criteria. The pairwise comparisons were calculated using Vargha-Delaney's $\hat{A}_{12}$, including its 95 per cent confidence interval and effect size. Moreover, we applied the Mann-Whitney U test to calculate the p-value of the effect size. Finally, this p-value was adjusted according to the Bonferroni correction. The results can be found in Table 5.2, 5.3, 5.4 and 5.5 for `CODE`, `NAME`, `SIMP` and `EXEC` respectively.

Feature coverage							
Pairwise comp.	p-val.	Adj. p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
JAC x LVS	5.060E-11	3.012E-10	0.444	[0.428, 0.461]	LVS	S	Yes
JAC x NCD	2.200E-16	1.320E-15	0.835	[0.821, 0.848]	JAC	L	Yes
JAC x RDM	2.200E-16	1.320E-15	0.680	[0.663, 0.697]	JAC	L	Yes
LVS x NCD	2.200E-16	1.320E-15	0.846	[0.833, 0.859]	LVS	L	Yes
LVS x RDM	2.200E-16	1.320E-15	0.701	[0.684, 0.716]	LVS	L	Yes
NCD x RDM	2.200E-16	1.320E-15	0.222	[0.208, 0.237]	RDM	L	Yes
Fault coverage							
Pairwise comp.	p-val.	Adj. p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
JAC x LVS	0.39	1	0.492	[0.475, 0.51]	LVS	S	No
JAC x NCD	2.200E-16	1.320E-15	0.607	[0.589, 0.624]	JAC	M	Yes
JAC x RDM	2.200E-16	1.320E-15	0.424	[0.407, 0.442]	RDM	S	Yes
LVS x NCD	2.200E-16	1.320E-15	0.622	[0.604, 0.639]	LVS	M	Yes
LVS x RDM	4.643E-08	2.784E-07	0.451	[0.434, 0.469]	RDM	S	Yes
NCD x RDM	2.200E-16	1.320E-15	0.328	[0.312, 0.345]	RDM	L	Yes

Table 5.2: Summary of the statistical analysis of the CODE criteria for both feature and fault coverage. Each pairwise comparison determines if there is statistically significant difference (SSD*) according to their p-values, Bonferroni adjusted p-values and the effect size (S = small, M = medium and L = large). Best corresponds to which of the two treatments are best according to $\hat{A}_{12}$.

As seen in the table above (Table 5.2) with the CODE criteria, the p-value and the adjusted p-value is always below our α set at 0.05. This means that there is statistical evidence that drawing a random value from each of the compared samples means that the two values will not be the same. However, for fault coverage there is not enough statistical evidence to draw that conclusion for JAC and LVS. This is supported by $\hat{A}_{12}$ which is very close to 0.5. Note that for the CODE criteria (Table 5.2) and feature coverage, LVS performs better than RDM, but for fault coverage, it is the other way around. Moreover, the effect size between RDM and JAC is large for feature coverage, but only small when it comes to fault coverage.

If we instead look at the NAME criteria in Table 5.3, all effects results in a statistically significant difference. For feature coverage, all effect sizes related to NCD are the only ones that are large, which is confirmed by the visual analysis (Figure 5.3a). Another thing to note concerns fault coverage data where the effects between JAC, LVS and RDM are small, and this is also confirmed by our visual analysis (Figure 5.2a).

Feature coverage							
Pairwise comp.	p-val.	Adj. p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
JAC x LVS	2.200E-16	1.320E-15	0.425	[0.408, 0.443]	LVS	S	Yes
JAC x NCD	2.200E-16	1.320E-15	0.816	[0.802, 0.830]	JAC	L	Yes
JAC x RDM	2.130E-05	1.290E-04	0.538	[0.520, 0.556]	JAC	S	Yes
LVS x NCD	2.200E-16	1.320E-15	0.827	[0.813, 0.840]	LVS	L	Yes
LVS x RDM	2.200E-16	1.320E-15	0.606	[0.589, 0.623]	LVS	M	Yes
NCD x RDM	2.200E-16	1.320E-15	0.224	[0.210, 0.239]	RDM	L	Yes
Fault coverage							
Pairwise comp.	p-val.	Adj. p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
JAC x LVS	2.200E-16	1.320E-15	0.417	[0.400, 0.434]	LVS	S	Yes
JAC x NCD	2.200E-16	1.320E-15	0.659	[0.642, 0.675]	JAC	M	Yes
JAC x RDM	1.410E-03	8.460E-03	0.472	[0.454, 0.489]	RDM	S	Yes
LVS x NCD	2.200E-16	1.320E-15	0.714	[0.698, 0.729]	LVS	L	Yes
LVS x RDM	1.154E-13	6.924E-13	0.566	[0.549, 0.584]	LVS	S	Yes
NCD x RDM	1.154E-13	6.924E-13	0.332	[0.316, 0.349]	RDM	M	Yes

Table 5.3: Summary of the statistical analysis of the NAME criteria for both feature and fault coverage. Each pairwise comparison determines if there is statistically significant difference (SSD*) according to their p-values, Bonferroni adjusted p-values and the effect size (S = small, M = medium and L = large). Best corresponds to which of the two treatments are best according to $\hat{A}_{12}$.

Feature coverage							
Pairwise comp.	p-val.	Adj. p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
JAC x LVS	4.210E-14	2.526E-13	0.434	[0.416, 0.452]	LVS	S	Yes
JAC x NCD	2.200E-16	1.320E-15	0.798	[0.783, 0.812]	JAC	L	Yes
JAC x RDM	3.440E-04	2.064E-03	0.532	[0.514, 0.549]	JAC	S	Yes
LVS x NCD	2.200E-16	1.320E-15	0.808	[0.794, 0.822]	LVS	L	Yes
LVS x RDM	2.200E-16	1.320E-15	0.587	[0.570, 0.604]	LVS	S	Yes
NCD x RDM	2.200E-16	1.320E-15	0.222	[0.208, 0.237]	RDM	L	Yes
Fault coverage							
Pairwise comp.	p-val.	Adj. p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
JAC x LVS	2.200E-16	1.320E-15	0.388	[0.371, 0.406]	LVS	M	Yes
JAC x NCD	2.200E-16	1.320E-15	0.591	[0.573, 0.608]	JAC	S	Yes
JAC x RDM	1.096E-15	6.580E-15	0.429	[0.412, 0.446]	RDM	S	Yes
LVS x NCD	2.200E-16	1.320E-15	0.686	[0.670, 0.702]	LVS	L	Yes
LVS x RDM	4.190E-11	2.510E-10	0.556	[0.542, 0.576]	LVS	S	Yes
NCD x RDM	2.200E-16	1.320E-15	0.344	[0.327, 0.361]	RDM	M	Yes

Table 5.4: Summary of the statistical analysis of the SIMP criteria for both feature and fault coverage. Each pairwise comparison determines if there is statistically significant difference (SSD*) according to their p-values, Bonferroni adjusted p-values and the effect size (S = small, M = medium and L = large). Best corresponds to which of the two treatments are best according to $\hat{A}_{12}$.

As with the NAME criteria in Table 5.3, all pairwise comparisons for the SIMP criteria

in Table 5.4 results in statistically significant differences. For feature coverage, the effect of removing the ECU portion of test case names (i.e. the difference between the SIMP and NAME criteria) results in LVS performs more equal to RDM , which match the results from our visual analysis. On the other hand, for fault coverage, JAC has become more similar to NCD, thus reducing the effect size.

Table 5.5 shows the EXEC criteria. It is not a surprise given the results from the visual analysis (see Figure 5.2d) that the effect size is large in favour of XOR with regards to fault coverage. However, it is also not surprising that RDM is better with regards to feature coverage, due to the results of the visual analysis (see Figure 5.3d).

Feature coverage						
Pairwise comp.	p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
XOR x RDM	2.200E-16	0.405	[0.388, 0.422]	RDM	S	Yes
Fault coverage						
Pairwise comp.	p-val.	$\hat{A}_{12}$	Effect size (CI)	Best	Effect size	SSD*
XOR x RDM	2.200E-16	0.862	[0.850, 0.874]	XOR	L	Yes

Table 5.5: Summary of the statistical analysis of the EXEC criteria for both feature and fault coverage. Each pairwise comparison determines if there is statistically significant difference (SSD*) according to their p-values, Bonferroni adjusted p-values and the effect size (S = small, M = medium and L = large). Best corresponds to which of the two treatments are best according to $\hat{A}_{12}$.

5.1.3 Execution time results

An important part of performing prioritisation is whether or not the time required for prioritisation and execution of a subset of test cases, that achieves acceptable coverage, actually is shorter than executing all test cases or randomly selecting test cases to run. Thus we performed a single measurement of each treatment for each criteria, except for RDM which was only measured once. The execution times does not include the time required for set-up and tear down of test cases and are thus approximations.

Table 5.6 presents the execution times and prioritisation times when prioritisation was performed using the CODE criteria. What might be more interesting here, without comparing execution times with that of coverage, is the execution times for NCD. These are always considerably lower than the other treatments. When it comes to JAC and LVS, they take considerably longer than RDM early in suite selection. It is not until 85 per cent of the suite is selected, that RDM catches up to similar timings as LVS and JAC.

CODE	RDM	JAC	LVS	NCD
Prep	0	0.11	0.10	0.14
5%	4.92	14.43	14.43	0.18
10%	10	20.55	22.31	0.32
15%	10.47	29.66	29.68	2.72
20%	15.88	32.48	32.48	5.85
25%	18.54	41.13	42.75	5.98
30%	25.21	45.21	42.96	6.13
35%	25.41	48.38	46.13	6.3
40%	25.82	48.51	48.5	6.43
45%	30.63	48.65	48.63	6.57
50%	31.27	48.8	48.78	6.72
55%	32.17	48.95	49.41	6.87
60%	33.92	49.1	51.31	7.02
65%	35.61	49.23	51.46	7.17
70%	43.16	50.88	51.63	7.33
75%	43.31	51.53	51.76	7.48
80%	45.74	51.92	51.91	7.62
85%	49.19	52.06	52.06	10.02
90%	49.85	52.21	52.21	26.92
95%	50	52.34	52.35	34.02
100%	52.51	52.51	52.51	52.51

Table 5.6: Approximate execution times in minutes for all treatments related to the CODE criteria. The *Prep* value is the time required by the technique to execute. The listed times does not include set-up and tear down processes.

EXEC	RDM	XOR
Prep	0	0.11
5%	4.92	0.15
10%	10	0.53
15%	10.47	15.13
20%	15.88	26.12
25%	18.54	31.17
30%	25.21	31.32
35%	25.41	31.47
40%	25.82	36.12
45%	30.63	42.01
50%	31.27	42.96
55%	32.17	43.11
60%	33.92	43.24
65%	35.61	43.39
70%	43.16	46.24
75%	43.31	48.51
80%	45.74	51.9
85%	49.19	52.05
90%	49.85	52.2
95%	50	52.35
100%	52.51	52.51

Table 5.7: Approximate execution times in minutes for all treatments related to the EXEC. The *Prep* value is the time required by the technique to execute. The listed times does not include set-up and tear down processes.

When it comes to Table 5.7, we can see the execution and prioritisation times of the EXEC criteria. Note that XOR achieved 100% fault coverage earlier than the other techniques requiring only 20% of the suite selection (Figure 5.2d), which results in roughly 26 minutes to achieve full fault coverage (Table 5.7). This can be compared

to RDM which has an execution time of almost 16 minutes, but only 20 per cent fault coverage (Table 5.7).

NAME	RDM	JAC	LVS	NCD
Prep	0	0.11	0.11	0.11
5%	4.92	14.56	13.36	0.18
10%	10	23.21	29.36	0.32
15%	10.47	33.27	37.70	2.72
20%	15.88	38.5	41.06	5.85
25%	18.54	40.26	42.83	5.98
30%	25.21	40.40	45.23	6.13
35%	25.41	42.86	45.4	6.3
40%	25.82	43.81	46.28	6.43
45%	30.63	46.31	47.18	6.57
50%	31.27	46.46	48.33	6.72
55%	32.17	46.61	48.48	6.87
60%	33.92	47.26	49.11	7.02
65%	35.61	48.16	49.51	7.17
70%	43.16	49.08	50.41	7.33
75%	43.31	50.83	51.53	7.48
80%	45.74	50.98	51.91	8.12
85%	49.19	51.36	52.06	10.02
90%	49.85	51.5	52.21	28.63
95%	50	52.38	52.36	34.29
100%	52.51	52.51	52.51	52.51

Table 5.8: Approximate execution times in minutes for all treatments related to the NAME criteria. The *Prep* value is the time required by the technique to execute. The listed times does not include set-up and tear down processes.

SIMP	RDM	JAC	LVS	NCD
Prep	0	0.10	0.11	0.14
5%	4.92	0.42	5.71	0.18
10%	10	4.72	9.51	0.32
15%	10.47	4.87	16.21	2.72
20%	15.88	7.66	22.88	5.85
25%	18.54	18.79	31.43	5.98
30%	25.21	29.37	32.58	6.13
35%	25.41	30.29	32.33	6.3
40%	25.82	32	33.88	6.43
45%	30.63	35.68	35.28	6.57
50%	31.27	36.37	36.18	6.72
55%	32.17	45.03	36.58	6.87
60%	33.92	47.51	37.45	7.02
65%	35.61	49.53	50.5	7.17
70%	43.16	50.2	50.88	7.33
75%	43.31	50.31	51.78	7.48
80%	45.74	50.96	51.93	7.62
85%	49.19	51.11	52.08	10.02
90%	49.85	52.25	52.21	26.92
95%	50	52.38	52.36	34.02
100%	52.51	52.51	52.51	52.51

Table 5.9: Approximate execution times in minutes for all names related to the SIMP criteria. The *Prep* value is the time required by the technique to execute. The listed times does not include set-up and tear down processes.

Moving on to Table 5.8 which presents the NAME criteria. Yet again we can see that JAC and LVS present very similar execution times. Meanwhile, NCD presents considerably lower execution times. For JAC the increase in execution time is quite

steady for each additional five percentage points. The same applies to `LVS`. `NCD` on the other hand, has a slowly increasing execution time, but when it selects 90% of the suite it increases its time with 18 minutes. The same happens when it goes from 95 to 100 per cent suite selection.

Table 5.9 presents the execution times for all the treatments of the `SIMP` criteria. Given the similarity of the `NAME` and `SIMP` criteria, it is interesting that `JAC` and `LVS` differ as much as they do in Table 5.9. `JAC` is always executed faster than `LVS` and at 25% suite selection, `JAC` takes almost 19 minutes to execute. Meanwhile, `LVS` requires 31 minutes to execute 25% of the prioritised list. This can be compared to Table 5.8, where the longest difference between the two is about 5 minutes.

5.2 Visualisation

The interview sessions and evaluation provided us with a lot of feedback regarding the quality of the visualisations and how they could be improved. Moreover, we learned from practitioners about what they think is required for using automated test optimisation. But also how automated test optimisation could affect them and their practices.

The thematic analysis performed on the qualitative data resulted in the thematic map as seen in Figure 5.4. It contains two themes: *Test process* and *Visualisation theme*. The test process only contains two codes and focuses on the test process employed by the subjects.

Decision information contains highlights on what information the subjects currently use and what would like to use. While *Consequences* identifies what they believe automated test prioritisation would affect them.

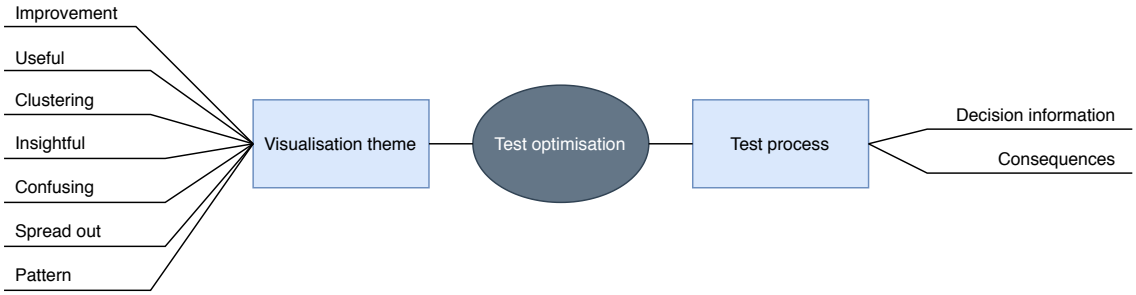


Figure 5.4: The thematic map based on the analysis results of our thematic analysis. It highlights the identified themes and codes found in the interview transcripts.

The Visualisation theme contains seven codes:

- *Improvement* - This code was put on everything that contained some kind of

suggestion on changes that should be made to our visualisations.

- *Useful* - The useful code was primarily used for the history plot to determine if the subjects considered it to be useful for them.
- *Clustering* - Clustering was used to highlight clusters and discussions about clusters for the similarity maps.
- *Confusing* - Everything that hampered or hindered the interviewees ability to draw conclusions on our visualisations was assigned this code.
- *Insightful* - Every mentioned insight was applied this code. We considered an insight to be reasoning and conclusions about patterns or areas of the visualisations.
- *Spread out* - Spread out is the opposite of the Clustering code. It is coded when the interviewees mention spread out test cases in the similarity maps.
- *Pattern* - This code was mostly used when interviewees identified patterns in the history plot.

5.2.1 Similarity maps

The majority of the participants were positive regarding the use and quality of the similarity map. Many participants also felt that it provided them with a holistic view of their testing repository. All six participants ranked t-SNE as more clustered compared to MDS. Moreover, the spread out nature of the test cases in the MDS visualisation prompted a participant hampered two participants understanding:

> *"But the general spread in this area I feel is so general that I would not really know if they are similar or not." - A*

Another participant put it another way:

> *"This graph [MDS] is a bit more difficult to read I think. [...] It is a lot of dots all over the place and it is only when they are grouped, like this, that you can get some information out of it." - C*

When it comes to drawing conclusions from the similarity maps, we had some problems catching this in our instrument. However, during some sessions this would be brought up by either the interviewee or the interviewer as a part of follow-up questions. For example, one participant presented a concrete example of his conclusion:

> *"That we are bad at naming the test cases" - F*

Another participant mentioned the usefulness of knowing how many test cases that are testing the same or similar areas and features. Since the visualisations were

based on `NAME`, two participants mentioned that it was clear why some test cases were clustered together in the t-SNE visualisation:

> *"But it makes sense that all these are grouped together because they are very similar in the test case name." - C*

On the other hand, everything was not that clear for the participants and many issues with the maps were presented. Primarily many of the participants saw the spread of test cases in MDS as a problem that hinders them from drawing conclusions on the similarity between test cases. Another participant thought that it was difficult to connect their own knowledge about the tests to what they say in the maps. One participant highlighted the need of deep understanding of the domain (both with regards to the domain itself, but also the test cases in the map) to be able to fully understand and interpret the similarity maps.

> *"[The similarity map] is more into the classification of test cases and that would require deep understanding of the domain level." - D*

When it comes to improvements of the similarity maps, the most prominent problem is understanding why some test cases are similar to each other. One suggestion was to implement support for regular expressions to highlight areas in the maps where test cases which matching names could be found. Another idea was to show why some test cases are clustered when the user hover over clusters. A third suggestion was to add some kind of relational map next to the similarity map, so that it better showcase relationships between both test cases and clusters.

5.2.2 History plot

Our own creation, the history plot, had a warm reception by the interviewees and all of them believed that they could make use of it.

> *"We have bugs, will get the bugs on a regular basis. So this chart will help us in analysing the bugs. Like, how many test cases affected with similar issues? How that trend is for those kind of issues." - F*

Multiple participants mentioned that the plot would help them with following changes and statistics over time and thus enable them to more broadly identify issues. These issues could be anything from the testing environment, to the hardware and the software itself. Some of the participants were able to draw conclusions from what they identified in the plots:

> *"[...] and these specific test cases, it is because we can see that this is probably one of the ECUs that fails." - A*

The majority of the participants are able to identify patterns in the plot and to a certain degree discuss the reason why they appear. Additionally, they indicated

that the details presented in the history plot provided a holistic view regarding the health and quality of their test repository, but also of their testing environment.

Some suggestions were mentioned with regards to the history plot. The simplest suggestion was to change the x-axis to show dates instead of a execution run counter. Another suggestion was to show a pass/fail ration for each execution by adding another axis at the top of the plot. Meanwhile, a third suggestion was to show a percentage of pass versus failure for each test case on the right hand side of the plot.

5.2.3 Testing practices

An important part of testing is making decisions, both on which test cases should be developed next, but also deciding on overall guidelines for testing. Thus we asked the participants what information they based their decisions on. Many of them said that previous experience and information on workload, time frames and project phase where vital to make good decisions. One participant mentioned test specifications which state what should be tested and what requirements applies. These specifications are created when new features need to be developed for the testing environment. Another important source in decision making are requirements:

> *"Basically, we will try to get the requirements. If requirements are good enough, then we will try to derive the use cases from them and we will try to test all the use cases on that." - E*

From the transcripts we were able to extract what some participants want to base decisions on. One interviewee thought that trends on bug fixing would be useful:

> *"[...] like to know the trends and fixing the defects." - E*

Another participant mentioned execution results from test cases to improve them. However, this is something that is already in use but something that could be improved by using the history plot visualisation.

One interviewee identified some issues that needs to be addressed before they could make use of automated test prioritisation. He believes that the overall coverage needs to improve before it becomes relevant to reduce the number of executed test cases:

> *"One of the main problems we have is to optimise our test coverage. That is something which is required I think before we move into like optimisation and reduce the number of test cases which will run as a part of [CI pipelines] [...]" - D*

Most participants do not think they have to change the way they work if they use automated prioritisation. However, they believe that they will decrease the length of the feedback cycle. For example, interviewee *A* believes that testers would receive feedback faster and thus will discuss the results more often with the developers:

"If automatisation was applied we would probably get faster feedback because they are running fewer tests and then they can get faster feedback on test cases, which they can ask us about. In that regard we would at least speed up the process." - A

6

Discussion and conclusion

Chapter 6 begins with a discussion and analysis of the results regarding the tool and the case study evaluation. Following up is the answers to the research questions posed earlier in this thesis. This chapter then moves on to discuss the relationship between our findings and related work before going into the possible threats to our validity in Section 6.2 and future research in Section 6.3. Finally, this thesis ends after presenting its conclusion in Section 6.4.

This study shows that there is significant statistical evidence for the `EXEC` criteria, together with the `XOR` distance measure to be a viable alternative to more common diversity measures (such as Levenshtein) and criteria. The study also shows that Levenshtein is a very viable option for static types of criteria, which previous research in the field supports.

What is new in this study is that it shows that `XOR`, together with the `EXEC` criteria, performs much better than other techniques when it comes to detecting faults. This can be explained by the usage of execution results in the `EXEC` criteria due to our assumption that each failure is a unique fault. It can also be explained by the close relationship that faults have to failures. However, it might be more probable that multiple failures discover the same fault, but it is not something we can discuss in this study. When it comes to the feature coverage measure, `XOR`, together with the `EXEC` criteria, does not perform equally well as it did in fault coverage. That result is expected since there is a very loose connection between features and execution results. This connection builds upon the assumption that if two test cases fail at the same time, then they might be testing the same feature. However, it is also possible that the reason for them failing is that they find the same fault. This fault does not necessarily have anything to do with the feature they are testing and might not even exist in the targeted feature.

Generally, the code contained in a test case has a strong relationship to the feature it tests. Thus it does not come as a surprise that treatments perform well for feature coverage. Moreover, the statistical analysis of `CODE` for feature coverage supports the conclusions that can be drawn from Figure 5.3c. If we compare the feature coverage for `CODE` with its fault coverage, we are not surprised that we are unable to find a statistically significant difference between `LVS` and `JAC`. Visually, the measures are similar in both Figure 5.3c and 5.2c, but the values differ on is many more for

feature coverage than for fault coverage. Thus, even with the larger magnitude in the difference between LVS and JAC, they have many more values which are equal to each other.

The variations in feature and fault coverage between NAME and SIMP can be explained by the difference in the underlying data. The difference between the NAME and the SIMP criteria is the removal of ECU information from test case names in SIMP. This removal results in test cases that had different names, now instead have the same names. For example we have the two test cases: *ABC - test boot* and *DEF - test boot*, both testing a boot operation, but on different ECUs (named *ABC* and *DEF*). In NAME, these test cases would not be identical, but for the SIMP criteria, they would be identical. Thus only one of them would be added to the prioritised list at different indices. This difference is also supported by the feature coverage plots (see Figure 5.3) where LVS is only slightly better than RDM for SIMP, compared to a medium effect for NAME.

For the CODE criteria, the fault and feature coverage differ a lot. This can be explained by the code of test cases having much more in common with what feature they evaluate, than what fault they discover. In other words, it is easier to identify different features from the test case code, than it is to identify different faults. Moreover, it is more difficult to identify identical test cases based on code since the code usually contains many more characters than their names.

Since CODE performs better than NAME with regards to feature coverage, LVS and JAC, it is reasonable to suggest that test case code has a closer relationship with features than test case names.

Even though NCD performs poorly when it comes to coverage and APFD, it is in some scenarios more worth to use NCD because of its low execution times, even after selecting most of the prioritised list. However, the observed execution times for NCD is so low that there might be other reasons behind these values. For example, it could be explained by not making multiple calculations of execution times and this applies to the other execution times as well. In some situations, it is often more effective to randomise the test cases' order than to actually use NCD. What we can see is that NCD requires longer time to create the diversity matrix the longer the data is for the criteria and the number of test cases. Thus, NCD will take significantly longer time if the test repository contains large amounts of test cases.

When it comes to the performance of XOR and EXEC we did suspect that its feature coverage would be low because of the loose connection between execution history and features. There is not enough information in execution history for XOR to be able to discern any differences between features. As compared to the other criteria in this study which all have some sort of relationship with features. For example, test case names could describe which feature the test evaluates, while the code describes how it should run, thus we can identify features by analysing method calls.

RQ1 - How can we instrument CI pipelines to optimise test feedback

cycles? We have shown that it is possible to equip CI pipelines with support for automated test optimisation which would enable faster feedback cycles. The description of our architecture and tool support (detailed in Chapter 3) comprise the main elements needed to instrument the workflow of diversity-based optimisation. Some interviewees expressed that the most significant change automated optimisation would provide was faster feedback. However, there might be need for further iterations of design, including evaluations, to further identify and specify the perfect instrumentation.

RQ1.1 - What different types of information can we use to optimise tests in CI pipelines? We have identified four potential criteria for use in test optimisation in the context of CI pipelines: `CODE`, `NAME`, `SIMP` and `EXEC`. However, `EXEC` shows the most promise with regards to fault detection, and `SIMP` can be hard to implement in other organisations due to differences in the encoding of test cases. We believe that the dynamic information type has the highest potential but requires further research. Particularly, the choice between both could be situational (i.e., the tester decides depending on the testing goal - feature or fault coverage), or create a hybrid technique able to incorporate both information types as a multi-objective optimisation.

RQ1.2 - What are the trade-off in using different types of information from test artefacts? With regards to dynamic information, it quickly reaches high fault coverage but it will struggle to reach a reasonable amount of feature coverage. With 100 per cent fault coverage, the execution history based `XOR` algorithm executes only 20 per cent of the full suite and it takes only about half the time required to run the entire test suite.

The static information type often require long test execution times when it has high fault or feature coverage. Some distance measures and criteria have shorter execution times with a majority of the suite selected, but it comes at a cost of longer time for calculating the distances. By using Normalised Compression Distance the execution time decreases drastically, but so does the coverage. However, at between 80 and 95% suite selection, the coverage from `NCD` is as high as other treatments, but its execution time is still considerably lower than the others.

RQ1.3 - How effective are the optimised test suites? When it comes to the static information, we are able to achieve high fault and feature coverage while also maintaining low execution times. This is due to using the Normalised Compression Distance. However, this goes against the results of our statistical analysis, but that analysis only looked at coverage data, it did not look for correlations between coverage and execution times.

For the dynamic criteria, i.e. execution history, the `XOR` distance measure performs significantly better than the random control group when it comes to fault detection. However, it is not as good as `RDM` when it comes to feature coverage, even if the difference is much smaller.

RQ2 - How can we use test optimisation to support human decision making in test cycles? The most important information extracted from test optimisation can be visualised in *similarity maps*. These maps allow practitioners to extract information about the test repository that they previously has not been able to analyse. Secondly, a *history plot* based on execution history (as used for prioritisation with XOR distance measure), supports practitioners responsible for the testing environment, as well as developers and testers in making correct decisions about the status of the software and the test repository.

RQ2.1 - What type of information do stakeholders use to make testing decisions? Decisions made by interviewed subjects are based on a wide range of information. This includes requirements, test specifications and their own experience as bases for decisions. These differences between information is not surprising since requirements often place a big role in large software projects. Often the deciding factor in testing decisions is the estimation of how long time it will take to integrate a feature. It is an estimation that is only based on the practitioner's previous experience. Our research shows that practitioners would like to use trends on bug-fixing and that information would be more accessible with our history plot.

We suggest that decision information is divided into two categories. Administrative and testing information. Administrative information is, for example, how much time is required during different project phases and previous experience of development and testing. On the other hand, testing information regards test cases and execution data, for example, if the test cases have failed and test specifications.

What is interesting about these two categories is that both are relevant when making testing decisions. It is essential to find a balance between them in a way that fits the situation. For example, when discussing if a test specification should be implemented during the current sprint or the next, it is important to understand the test specification and be able to present an estimate on the implementation time.

RQ2.2 - To what extent can we capture and visualise this information to trigger insights from testing cycles? Information can be captured using two types of plots: Similarity maps and history plot. Most participants could relate to and understand similarity maps based on t-SNE and they had more trouble understanding MDS because of its spread. We suggest using t-SNE since it allows practitioners to draw conclusions about how they work with tests and the quality of their test repositories. An issue with the similarity maps was that we were required to thoroughly explain what it represents before they were able to interpret it. Overall, the similarity maps provide practitioners with a holistic view of their test repository.

Most participants were able to draw conclusions based on the history plot and each participant seemed to draw different insights from the same identified patterns. When practitioners are able to draw different conclusions from the same data it can enable better discussions and test cases if the visualisation is used properly. Our overall impression is that participants found the history plot to be more helpful than

the similarity maps. This is supported by our impression that the history plot not only gave participants a holistic view of the health and quality of the test repository, but also of the testing environment.

RQ3 - How does the integration of test optimisation in CI pipelines affect practitioner's feedback cycles? Our research show that the use of test optimisation in CI pipelines would enable faster feedback cycles and could therefore result in higher workload for developers. It could potentially shorten the length of the test cycle itself which in turn could shorten time to market. The visualisation that were create from the use of test optimisation can help developers and testers when maintaining and improving the quality of test repositories and test environments.

The generalisability of our result varies. Our results regarding the visualisations (similarity maps and history plots) are generalisable since they are not based on information only available at the case company. Moreover, the architecture of the tool and the tool itself is generic due to its modularity and the description of how it was developed, which others are free to use. The results on the execution history criteria can be general, but requires more research. On the other hand, the `SIMP` criteria is unique for the case company due to the specific removal of the ECU from test case names. Other organisations can probably use the test case name criteria; however, its effectiveness (as seen in this study) will most likely vary. We believe the use of test case code as a criteria is possible for any organisation to use, but as with test case names, the effectiveness will vary.

6.1 Related work

An earlier discussion mentioned the low performance of `NCD` in our study, with regards to coverage, and this result contradicts previous research. Previous research has shown that NCD achieves higher coverage and has shown great potential [4], [7]. There are a number of reasons for our results for `NCD`, the first one regards the length of the strings. The compared strings used in NCD should be long for the compression algorithms to clearly identify the differences. If a very short string is sent into a compression algorithm it might come out being longer, through the addition of a head an tail containing information on, e.g.,how to properly decompress the string. Thus, if we have two strings: "a" and "b", they will most likely be identical according to NCD since they are so short. However, this should have resulted in `NCD` performing better in `CODE` than `NAME` or `SIMP`, but it does not which is very surprising. Another possible reason for our result lies in the library we use to calculate the NCD. It might be so that the library uses a compression algorithm that is a bad approximation of Kolmogorovs complexity. This is closely related to the third reason which could be that the library used for calculating NCD does not remove the header from the compressed string, which would affect the resulting distances.

When it comes to `JAC` and `LVS`, they perform mostly equally well regarding their coverage in our study. For example, de Oliveira Neto *et al.* [42] has shown that

Levenshtein and Jaccard Index perform similarly well when it comes to coverage. However, they looked at coverage of requirements, dependencies and test steps, while we look at fault and feature coverage. While we have not achieved identical coverage results (since we look at different types of coverage), the differences between the individual diversity measures are similar.

Since previous research in the field of test optimisation has not focused on the dynamic type of information, our findings regarding the use of execution history in test prioritisation is novel.

There is a paper from de Oliveira Neto *et al.* [9] that presents how similarity maps based on MDS can visually support practitioners. Their research is something that we extend by investigating how good MDS and t-SNE are for reducing the dimensions of diversity matrices to visualise them in similarity maps. This thesis has also examined why the algorithms are good and if the similarity maps are insightful or not for practitioners. Moreover, we extended the conclusions presented by de Oliveira Neto *et al.* by performing the study in a different domain, namely automotive domain. When it comes to our research on the history plot, the plot has shown to be insightful for practitioners. Thus, it is novel even if there already exists similar plots already as plug-ins for build systems[1].

6.2 Threats to validity

Internal validity

A threat to our internal validity regards the possibility of our interview instrument not fully capturing all aspects of our research questions. However, it was mitigated by iterating the interview form with an experienced third party. Additionally, our tool may suffer from a defective toolchain, meaning that all parts of it may not work as intended, which is also a treat to our internal validity. Since we ensured that each part of the tool operates correctly by performing unit tests, thus we mitigated this threat.

External validity

Since convenience sampling was conducted in our case study so that our interviews were performed with just one team of developers within a single company, our ability to draw any generalisable conclusions to the wider population is limited. That is not a major threat to our study, since both design science and case study tend to be bound by a limited context (i.e., a case company in our case). The findings should lead to future experimental studies on diversity-based optimisation in other contexts.

The first unit of analysis is also subject to the *interaction of selection and treatment*

[1] Test Results Analyzer - https://wiki.jenkins.io/display/JENKINS/Test+Results+Analyzer+Plugin

validity threat, since the analysed project only executes 183 test cases and has an abnormal pass and failure rate. Moreover, it is a project which is heavily tied to hardware which many software projects might not be. Therefore it hinder our ability to generalise our findings.

Construct validity

There are several threats to our construct validity in our study. The first threat is that there exists measurement biases in the first unit of analysis because a single type of measures and observations are used for some of our measures. For feature coverage, we do not have another measure that can be used to correlate these measures with each other. However, the first threat is partially mitigated for feature coverage due to having multiple observations in its data set. This makes it possible for us to identify other problems within the measure, such as odd values. Similarly with the execution time measure, it does neither have data points with multiple observations, nor another measure to be used for cross-checking. Therefore, this measure has most measurement bias. On the other hand, when it comes to fault coverage, we use this measure together with APFD to be able to cross-check them against each other. Thus, the first threat does not affect fault coverage. The second threat to our construct validity is about our participants trying to understand what our study is about and guess answers based on how they think we want them to behave during the interview session. We mitigated this threat by briefly describing the purpose of the interview without revealing too much information about our study.

Another threat to our construct validity is evaluation apprehension. During the interviews, the participants might have skewed their answers to make them look better. This can affect our results, but it is not something we can compensate for. Additionally, during the creation of the interview instrument an experienced third party evaluated the instrument. The purpose of this was to mitigate conscious and unconscious bias based on what outcomes we expected. Our first unit of analysis is also suffered from using APFD with failures instead of actual faults due to not having any information about defects from the test runs. This results in an additional threat to construct validity, which is difficult to mitigate. Although, since other researchers have done this simplification in their research in the past [9], this is an acceptable practice. The same threat also applies to fault coverage because it is closely related to APFD.

A threat introduced by using Vargha-Delaney is that the effect size has the potential of being misinterpreted. This is caused by applying Vargha-Delaney on untransformed data [43]. We do not need to transform the data since the data was gathered for features and faults, both of which are relevant for stakeholders, since they convey the proportion of coverage (ranging from none to full coverage). Both ends of this spectrum, as well as the values within, are relevant to practitioners so they can understand how to detect minor changes in both types of coverage.

Conclusion validity

We have an increased threat of low statistical power due to using Bonferroni correction to mitigate the threat posed by our error rate. The low statistical power is usually an effect common for correction methods based on family-wise error rate.

The threat of violating assumptions of statistical threats was mitigated by using Kruskal-Wallis as statistical test since it is non parametric and the test is more conservative and less constrained by assumptions regarding the distribution of the data (particularly if compared to an ANOVA).

Another conclusion threat is the visual analysis of descriptive statistics. Unfortunately, we are not able to mitigate this threat at this point, and we hope future work can address it.

Moreover, we have a threat to our conclusion validity due to the reliability of tour measures. We mitigate this threat to the qualitative analysis by performing a thematic analysis of the interview, and by discussing the codes and themes use in said analysis with an experienced third party. This threat is also present for our fault coverage assumption. The assumption is that we consider each individual failure as a unique fault. Thus we say that something that might not be a fault, actually is classified as a fault. Unfortunately, we are unable to mitigate this threat since there is no alternative data available which could pin-point faults in the software under test.

The final threat to our conclusion validity is the reliability of treatment implementation. We mitigated this threat by using the same scripts and data (such as test cases and execution history) to make sure that no modifications are done to the source material.

6.3 Future research

Since this was just the first set of iterations of a design science study, there is a great potential in further development of the tool. There are suggestions that we received which could be added to the tool and evaluated to identify their value. Moreover, there are a lot of other criteria that could be evaluate (e.g., requirements and documentation). There are other diversity measure that could be evaluated. However, the most important future research from this thesis is combining dynamic criteria with static and evaluating weighting between them.

Another aspect to think of when extending this thesis is to have a wider variety and more participants in evaluations. This would enable more insights and a larger possibility of generalising the results. Moreover, it could be extended to include more than one case company and in different domains. Additionally, it would be more significant if other evaluation methods were used for the visualisation, e.g., focus groups and workshops.

When it comes to the history plot we believe there is potential in connecting executions to code changes. This was something this thesis did not look at since commits to the software repository did not trigger executions in the build system for the studied project. Therefore we were unable to look into this. It is something that future research should look into and see how it affects insights gained by practitioners.

6.4 Conclusion

This study contributes to the scientific knowledge by showing how effective execution history is with regards to finding faults. Our results show that traditional distance measures' ability to detect faults is nowhere near that of execution history and the XOR distance measure. However, static criteria far exceed their dynamic counterpart when it comes to feature coverage. Furthermore, we have shown that similarity maps and history plots can trigger insights among developers and testers about the quality of test repositories, testing practices, testing environment and the software quality. We have discovered that most interviewees gained insights from similarity maps using t-SNE. On the other hand, most participants struggled to gain insights on maps based on MDS. From the history plot, they were able to gain many insights on the quality of their testing cycles and the stability of their testing environment. Thus we believe that the use of similarity maps and history plots as parts of automated test prioritisation, increases the return of investment in these techniques.

We have also identified two categories of information which practitioners use in testing decisions: administrative and testing information. They are both critical to decision making and complement each other in different ways. However, further research is required to establish details of these categories.

The technical contribution from our research is a complete architecture and showcasing of libraries that are available to practitioners and organisations to implement their take on test optimisation in their CI environment.

Bibliography

[1] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances", *Automated Software Engineering*, vol. 19, no. 1, pp. 65–95, 2012.

[2] Y. Ledru, P. Alexandre, and S. Boroday, "Using string distances for test case prioritisation", in *International Conference on Automated Software Engineering*, ser. 2009 IEEE/ACM, IEEE Computer Society, 2009, pp. 510–514. DOI: 10.1109/ASE.2009.23.

[3] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto, "On the use of a similarity function for test case selection in the context of model-based testing", *Software Testing, Verification and Reliability*, vol. 21, no. 2, pp. 75–100, 2011. DOI: 10.1002/stvr.413. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.413.

[4] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization", in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: ACM, 2018, pp. 222–232, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180210.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study", in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Aug. 1999, pp. 179–188. DOI: 10.1109/ICSM.1999.792604.

[6] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics", *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008. DOI: 10.1109/icstw.2008.36.

[7] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases", *CoRR*, vol. abs/1506.03482, 2015. arXiv: 1506.03482. [Online]. Available: http://arxiv.org/abs/1506.03482.

[8] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization", in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 523–534. DOI: 10.1145/2884781.2884791.

[9] F. Gomes de Oliveira Neto, R. Feldt, L. Erlenhov, and J. Benardi de Souza Nunes, "Visualizing test diversity to support test optimisation", *arXiv e-*

prints, arXiv:1807.05593, arXiv:1807.05593, Jul. 2018. arXiv: 1807.05593 [cs.SE].

[10] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration improving software quality and reducing risk*. Addison-Wesley, 2007.

[11] D. Farley and J. Humble, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[12] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity", *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, 6:1–6:42, Mar. 2013, ISSN: 1049-331X. DOI: 10.1145/2430536.2430540.

[13] R. Cilibrasi and P. Vitanyi, "Clustering by compression", *IEEE Transactions on Information Theory*, vol. 51, no. 4, pp. 1523–1545, 2005. DOI: 10.1109/tit.2005.844059.

[14] P. Jaccard, "Etude de la distribution florale dans une portion des alpes et du jura", *Bulletin de la Societe Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, Jan. 1901. DOI: 10.5169/seals-266450.

[15] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals", *Soviet Physics Doklady*, vol. 10, pp. 707–710, 8 Feb. 1966.

[16] C. Bennett, P. Gacs, M. Li, P. Vitanyi, and W. Zurek, "Information distance", *IEEE Transactions on Information Theory*, vol. 44, no. 4, pp. 1407–1423, 1998. DOI: 10.1109/18.681318.

[17] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013.

[18] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne", *Journal of machine learning research*, vol. 9, pp. 2579–2605, Nov. 2008.

[19] W. S. Torgerson, "Multidimensional scaling: I. theory and method", *Psychometrika*, vol. 17, no. 4, pp. 401–419, Dec. 1952. DOI: 10.1007/bf02288916.

[20] K. Pearson, "On lines and planes of closest fit to systems of points in space", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901. DOI: 10.1080/14786440109462720.

[21] H. Hotelling, "Analysis of a complex of statistical variables into principal components.", *Journal of Educational Psychology*, vol. 24, no. 7, pp. 498–520, Oct. 1933. DOI: 10.1037/h0070888.

[22] L. Van Der Maaten, E. Postma, and J. Van den Herik, "Dimensionality reduction: A comparative review", *J Mach Learn Res*, vol. 10, pp. 66–71, 2009.

[23] S. V.S and S. Surendran, "A review of various linear and non linear dimensionality reduction techniques", *International Journal of Computer Science and Information Technologies*, vol. 6, pp. 2354–2360, 3 2015.

[24] G. J. Goodhill, M. W. Simmen, and D. J. Willshaw, "An evaluation of the use of multidimensional scaling for understanding brain connectivity", *Philosophical Transactions of the Royal Society, Series B*, vol. 348, pp. 265–280, 1994.

[25] G. E. Hinton and S. T. Roweis, "Stochastic neighbor embedding", in *Advances in neural information processing systems*, 2003, pp. 857–864.

[26] J. Cook, I. Sutskever, A. Mnih, and G. Hinton, "Visualizing similarity data with a mixture of maps", in *Artificial Intelligence and Statistics*, 2007, pp. 67–74.

[27] Y. Bengio, "Learning deep architectures for ai", *Foundations and Trends® in Machine Learning*, vol. 2, pp. 1–127, 1 2009.

[28] E. Engström, M.-A. Storey, P. Runeson, M. Höst, and M. T. Baldassarre, "A review of software engineering research from a design science perspective", *arXiv preprint arXiv:1904.12742*, 2019.

[29] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research", *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007.

[30] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Dec. 2008. DOI: 10.1007/s10664-008-9102-8.

[31] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies", *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, Feb. 2002, ISSN: 0098-5589. DOI: 10.1109/32.988497.

[32] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey", *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[33] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)", *Biometrika*, vol. 52, no. 3/4, p. 591, 1965. DOI: 10.2307/2333709.

[34] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis", *Journal of the American Statistical Association*, vol. 47, no. 260, p. 583, 1952. DOI: 10.2307/2280779.

[35] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other", *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. DOI: 10.1214/aoms/1177730491.

[36] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong", *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000. DOI: 10.3102/10769986025002101.

[37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[38] H. Lam, E. Bertini, P. Isenberg, C. Plaisant, and S. Carpendale, "Empirical studies in information visualization: Seven scenarios", *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 9, pp. 1520–1536, 2012. DOI: 10.1109/tvcg.2011.279.

[39] R. Mazza and A. Berre, "Focus group methodology for evaluating information visualization techniques and tools", *2007 11th International Conference Information Visualization (IV 07)*, 2007. DOI: 10.1109/iv.2007.51.

[40] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering", *2011 International Symposium on Empirical Software*

Engineering and Measurement, pp. 275–284, Sep. 2011. DOI: 10.1109/esem.2011.36.

[41] V. Braun and V. Clarke, "Using thematic analysis in psychology", *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006. DOI: 10.1191/1478088706qp063oa.

[42] F. G. de Oliveira Neto, A. Ahmad, O. Leifler, K. Sandahl, and E. Enoiu, "Improving continuous integration with similarity-based test case selection", in *Proceedings of the 13th International Workshop on Automation of Software Test*, ser. AST '18, Gothenburg, Sweden: ACM, 2018, pp. 39–45, ISBN: 978-1-4503-5743-2. DOI: 10.1145/3194733.3194744. [Online]. Available: http://doi.acm.org/10.1145/3194733.3194744.

[43] G. Neumann, M. Harman, and S. Poulding, "Transformed vargha-delaney effect size", vol. 9275, Sep. 2015. DOI: 10.1007/978-3-319-22183-0_29.

A

Interview instrument

The questions related to RQ2.2 are supported by research in the visualisation field [38], [39].

Time: approx. 30 minutes

This interview is part of our thesis and for this, we are interested in input from practitioners. Our thesis is about prioritising test cases but the focus of this interview is not on automated prioritisation, but instead on how we can involve practitioners in the CI pipeline.

Consent for participation in the study:

Consent for an audio recording of this session:

Introductory questions

1. What is your current job title?
2. What do you do as a part of your work?
3. How long have you worked with software development and testing?

RQ2.2: To what extent can we capture and visualise this information to trigger insights from testing cycles?

This is a similarity map which displays how similar the test cases are to each other. The numerical distances are not relevant, which is why there are no values on the axes. We will show you two different similarity maps and then a plot of test execution history.

Let them use the tool for about 1 minute then ask the following questions on Similarity maps:

1. Do you notice any difference between the two plots? If so, what?
2. Can you identify any clusters of test cases?
3. Do you feel that this visualisation allows you to have a clear picture of the relations between test cases? How could it be improved?

The history plot shows execution results for each test case over time. Each square presets the result from a run of a specific test case, where green means that the test case passed. Red represents failure and white means that the test case wasn't executed.

Let them play with the plot for 1 minute.

1. Can you identify any recurring patterns in this visualisation? What is the reason behind this?
2. Can you see any use of this visualisation in your work?

RQ2.1: What type of information do stakeholders use to make testing decisions?

1. Do you develop new tests? If so, how? Do you use any specific process?
2. Do you make testing decisions? If so, what information are these decisions based on?
3. Do you add, remove or update the tests in the test repository? If so, how often?

RQ2: How can we use test optimisation to support human decision making in test cycles? (If there's time)

1. Do you use any type of test optimisation? For instance, if you cannot run all tests, do you choose or prioritise fewer tests and execute those?
2. Do you think anything has to be changed regarding testing practices to enable effective use of automated test optimisation? If so, what and how?

That's it for us, (and the end of the recording). Thank you very much for your participation.