OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Roope Kallio

TOWARDS TEST SUITE OPTIMIZATION IN SOFTWARE COMPONENT TESTING BY FINDING AND ANALYSING REPEATS IN TEST TRACE

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2015

Kallio R. (2015) Towards Test Suite Optimization in Software Component Testing by Finding and Analysing Repeats in Test Trace. University of Oulu, Department of Computer Science and Engineering. Master's Thesis, 54 p.

ABSTRACT

Regression testing is often a very resource and time consuming activity with large scale software. However, regression tests may be run in continuous integration where everything should run fast in order to get quick feedback.

This thesis reviews the existing solutions to optimize regression test suites and test suites in general by looking at various existing heuristic, computational intelligence and other methods. In most cases, these methods are based on the fact that test suites may contain redundant test cases. They also require code coverage, fault detection capability or requirement coverage information of each test case in order to be implemented.

Additionally, the thesis studies ways to detect redundancy in test trace. It is also discussed whether or not this kind of redundancy information can be used for test suite optimization instead. A new concept, pair-wise supermaximal repeats is presented.

A case study is conducted in software component continuous integration testing environment in Nokia Networks. Main methods include looking for patterns in test trace manually in a plot representing the test trace as well as finding pair-wise supermaximal repeats in the test trace with an enhanced suffix array implementation developed in test driven development method. A new application was developed, MRS Finder. The repeats are then analyzed.

The study shows that the there can be found huge amount of repetition which makes analyzing all of it hard by hand. The repeats also tend to occur more in the setup or teardown phase of a test case rather than in the action or assertion phase of the test cases. The study shows that MRS Finder offers different optimization possibilities than originally was thought.

Keywords: continuous integration, redundancy, suffix array, heuristics, computational intelligence, regression testing

Kallio R. (2015) Kohti testisarjan optimointia ohjelmiston komponenttitestauksessa etsien ja analysoiden toistoja testien jäljissä. Oulun yliopisto, tietotekniikan osasto. Diplomityö, 54 s.

TIIVISTELMÄ

Regressiotestaus on usein hyvin resursseja ja aikaa kuluttavaa suurten ohjelmistojen kanssa. Regressiotestejä voidaan kuitenkin ajaa jatkuvassa integroinnissa, jossa senkin tulisi suoriutua nopeasti, jotta palauteaika pysyy lyhyenä.

Tämä diplomityö käy läpi olemassa olevia ratkaisuja regressiotestien ja yleisestikin testisarjojen optimointia. Näistä käsitellään muun muassa heuristisia ja laskennallisen älykkyyden menetelmiä. Useimmissa tapauksissa nämä metodit perustuvat siihen, että testisarjat voivat sisältää testitapauksia, joissa on keskinäistä redundanssia. Menetelmät vaativat myös tietoa jokaisen testitapauksen koodikattavuudesta, virheiden paljastuskyvystä tai vaatimuskattavuudesta, jotta ne voidaan toteuttaa käytännössä.

Tämä työ tutkii myös tapoja löytää redundanssia testien jäljistä. Työ pohtii, voitaisiinko tätä tietoa käyttää testisarjojen optimointiin edellä mainittujen tietojen sijaan. Tämä työ esittelee uuden käsitteen, parittaisen supermaksimaalisen toiston.

Tapaustutkimus suoritettiin Nokia Networksin systeemikomponenttitason jatkuvan integroinnin järjestelmässä. Tapaustutkimuksessa kehitettiin testivetoisella kehityksellä MRS Finder -ohjelmisto, joka etsii toistoja parannetusta suffiksitaulusta ja tuottaa diagrammin, joka edustaa testien jälkiä. Diagrammista etsitään toistoja manuaalisesti ja suffiksitaulusta löydettyjä toistoja analysoidaan.

Tutkimus näyttää, että testien jäljistä voidaan löytää suuri määrä toistoja, joka tekee analysoinnista vaikeaa käsin. Toistot esiintyvät useammin testi tapauksien alkuasetus (setup) ja poiskorjaus (teardown) –vaiheissa kuin toiminta- tai vakuutusvaiheissa (assertion). Tutkimus näyttää myös, että MRS Finderilla löydetään erilaisia testi tapauksien optimointimahdollisuuksia kuin aluksi ajateltiin.

Avainsanat: jatkuva integrointi, redundanssi, regressiotestaus, suffiksitaulu, heuristiikka, laskennallinen älykkyys

TABLE OF CONTENTS

FOREWORD

I would like to thank Nokia Networks for the opportunity to conduct this study as an employee with thesis worker status. I look forward to possibilities to continue the study onwards.

I would also like to thank Jorma Taramaa, my line manager at Nokia Networks, and Jaakko Jokela, scrum master of our scrum team. They gave me assistance and guidance for the topics of the thesis. I would also like to thank Nokia Networks employee Janne Kohvakka for coming up with the main topic for the thesis. He also pointed me in the right from the beginning and provided the basic XML-parser for output.xml which I later on extended. Another employee of Nokia Networks, Mikko Tikkakoski helped me with the analyzing the results of the develop tool, MRS Finder, with his knowledge of the case study software application and SCT. I would also like to thank him and the rest of the case study software development team who have helped me along the way.

I would also like to thank my supervisor at Oulu University, Dr. Tech. Mika Rautiainen for overall scientific guidance on the thesis as well as evaluating the thesis. Dr. Tech. Professor Tapio Seppänen acted as the the second examiner for the thesis.

Oulu, 11.5.2015

Roope Kallio

ABBREVIATIONS

RTS	Regression test selection
CI	Continuous integration
SCT	System/software component testing
RTO	Regression test optimization
NP-complete	Both non-deterministic polynomial-time complex and non-deterministic polynomial-time hard
HGS	A heuristic test suite minimization algorithm developed by Harrold, Gupta and Soffa
PSO	Particle swarm optimization
LCS	Longest common sequence
CS	Common sequence
LCP	Longest common prefix
BWT	Burrows and Wheeler transform
MRS Finder	Maximal Repeat Sequence Finder
OOP	Object-oriented programming
JSON	JavaScript Object Notation
XML	Extensible Markup Language
CSS	Cascading Style Sheets
SCSS	Sassy CSS
Sass	Syntastically Awesome StyleSheets
jQuery	JavaScript library for easier event handling, animations etc.
QUnit	JavaScript unit testing library
AJAX	Asynchronous JavaScript And XML
UI	User interface
CLI	Command-line interface

1. INTRODUCTION

1.1. Motivation, research questions, scope and structure

In continuous integration, tests are run after each commit to verify the changes in the developed application. Often all functionalities are re-tested which we call regression testing. This kind of testing may take quite a lot of time and resources to run. The aim of this thesis was to study ways how we could go around this problem. It was suggested that test case traces should be investigated and looked for any redundancy. If there were any redundancy, some test cases can be removed. In order to find the answers to these questions I investigated several methods of test suite optimization and redundancy detection algorithms. In short, the main research question was: could a test suite be optimized using test trace sequence repetition detection and analysis.

During the literature research, an application, MRS Finder (Maximal Repeated Sequence Finder), was developed to find supermaximal and pair-wise supermaximal repeats in the test trace. The considered repeats are traced message sequences which were sent and received during the tests. To browse the results, an HTML5 application was developed into the MRS Finder. These were developed with test driven development method.

To set the scope of this thesis it was decided that different variations and specifics of the commonly used test suite optimization methods would be left out and we would concentrate on overviewing the methods. In addition, different redundancy detection algorithms were investigated but the study concentrates mainly on the suffix array method which was used in the python application. Other key points that this thesis reviews are optimization problems of the algorithms and their time and space complexity. All of this was done in order to find a well suited representation of the problem and an efficient algorithm to solve it. Different software testing methods and case study software application specific concepts are considered out of the scope of this thesis.

The thesis begins with a more detailed view of the problem and some background information on related software testing concepts in chapter 1.2. In chapters 2.1 and 2.2 an overview of test suite optimization is described with special grip on methods and objective functions that have been used before. In chapter 2.3, repetition detection methods are described along with the problems that are solved with them. In chapter 3 the scientific background of the relevant methods to the application, namely suffix array and supermaximal repeats, are described. Additionally, new concept, pair-wise supermaximal repeat is introduced. In chapter 4.1 the case study is described in such detail that the optimization problem and application development can be understood. In chapter 4.2 the results are represented along with analysis on them. In chapter 5, implementation problems, it's compatibility to test suite optimization and future work are discussed. Finally, the thesis is concluded in chapter 6.

1.2. Background

In continuous integration (CI), a change in the software triggers a build. In the build, tests are run to check if the change broke anything. These tests should be run fully automatically and all of them should pass in order for the build to be successful. [1 p.40] One of the key points is that everyone commits their work daily into the same repository so integration of the work can be verified. [2]

There are many benefits of CI. In its core, it removes the hard integration phase that is done to fix integration errors and makes it a simple every day activity. However, there has been some discussion about the scalability of CI. It has been stated that integration build will take longer as the code base grows. [2] One reason for this is that large code base increases the size of the test suites [3]. Increased build time leads to longer feedback time for developers who wait for results from CI. From a manager's point of view, waiting for results before proceeding with work means spending unproductive time. [2] However, the problem is not just the waiting, it is also the exhaustive use of computing resources which are often limited [3]. Bringing in more developers to a project can amplify the problem. First of all, the code base is expected to grow faster with more developers. Secondly, more people are waiting for the results of CI. [2]

Regression testing is defined as retesting of software. It is run when the software is modified. The main benefit of regression testing is the ensuring that the functionality of the software has not changed in the tested parts. [4] Affections to unmodified code that are caused by modified code are called regression errors which are what one wants to find in regression testing. [5] Test suites for regression testing are often maintained for the whole lifetime of an application, even after the original development process [3]. The simplest approach in regression testing is to re-execute all test cases. However, due to this approach being very expensive and resource consuming with large test suites, one often has to limit the number of test cases in the regression test. This process is called regression test selection (RTS). [4] Regression testing is increasingly hard in CI because of its time constraints [6].

2. TEST SUITE OPTIMIZATION BY DETECTING REDUNDANCY

2.1. Overview

A test suite consists of tests which work together in a natural way. Tests in a test suite share often same high-level objectives. Often there is also some data that is shared between the tests. Thus, test suite can be called a logical collection of tests and they should be created for efficient test case execution. [1 p.24]

System component testing, software component testing or often just component testing (SCT) is used to verify the functionality of a software component. Other main objective is to search for defects in components. A component can be defined as a part of software that is separately testable in component testing. [1 p.37]

The following specification can be considered as a good starting point for the structure of a test case: IEEE 829 Standard: Test Case Specification Template [1 p.80]

- Test case specification identifier
- Output specification
- Test items
- Environmental needs
- Input specifications
- Special procedural requirements
- Intercase dependencies

IEEE 829 standard has been used as a basis of recent ISO 29119 standard. However, ISO 29119 has created a lot of controversy and is therefore not used in this thesis. [7, 8]

To simplify IEEE 829, it can be said that test cases have a certain structure which basically consists of input and output specification. We get an output from an action with certain input specification. This output is then asserted against the expected output specification. [1 p.77-80]

Like in the IEEE standard, test cases have to set the environment correctly. This is usually done before the actual test in a test setup [1 p.23]. On the other hand, test teardown is done after the test case to clean up the environment. Suite setups and teardowns can also be run before the setup of each test case and after the teardown of each test case correspondingly. [9]

By putting this simplified structure together, a single test case execution can be defined like this:

1. Suite setup
2. Test setup
3. Action
4. Assertion (output against the output specification)
5. Test teardown
6. Suite teardown

In this thesis, let's define test case tracing and test tracing how tracing is defined in Robot Framework 2.8.7 documentation. Trace is a detailed debugging level and tracing is used to log what test libraries are doing internally. [9] Thus, in the context of test cases, test case tracing means logging of test case execution. (Execution) traces can be defined as low-level output data of tracing as in (Amiar, A. et al 2013)

[10]. In the case study of this thesis, we consider traced messages as our test trace. These are explained in more detail in chapter 4.1.3.

In software development, debugging has been said to be a very time-consuming activity which requires a lot of work especially with embedded software [10]. This is caused by the difficulty of finding faults in software. In literature we talk about fault localization. Execution traces have been used to locate these faults either manually or even automatically. [11] However, because of the great amount of low level data, traces are often very large. This problem can be amplified in embedded software. [10]

Black-box testing is defined as testing software with some known input and output specification without knowing what happens in the actual code of the software. Black-box testing can be therefore referred to as functional testing. On the other hand, white-box testing is about structural testing where the implementation is known. For example in white-box testing, a loop in the software can be tested by running it variable amount of times. [1 p. 84-86]

In this thesis we are going to need some notations for sequences. Assume we have a n-length sequence $S = \{s_1, s_2, \dots, s_n\}$ and 3-length sequence $S_{example} = \{'a', 'b', 'c'\}$. An ith prefix of S is a sequence $S_{pi} = \{s_1, \dots, s_i\}$ [12 p.392], in other words, a beginning of S. Similarly, an ith suffix of S is a sequence $S_{si} = \{s_i, \dots, s_n\}$ [12 p.986], an ending of S. A trivial case, empty sequence is considered both a suffix and prefix of all sequences [12 p.986]. To differentiate sequences from sets we will represent sequences as strings, for example $S_{example} = 'abc'$. Then, ith letter of string S can be represented with $S[i]$ and for example $S_{example}[0] = 'a'$ and $S_{example}[2] = 'c'$.

Typically in optimization, the problem is about finding the shortest distance between two points on a surface. Commonly there are also multiple objectives which cannot be achieved together. In order to perform better towards an objective one must sacrifice the performance towards another. Because of this, optimization is usually about decision making, treating nonlinear constraints and minimizing the objective function. [13] A solution to optimization problems can be sometimes called *an* optimal solution rather than *the* optimal solution because there may be multiple solutions that achieve the optimal value [12 p.359].

Test suites change over time as the application changes [14]. New test cases are added to a test suite in order to test a new feature that existing test suite did not test previously. This causes issues when old test cases are not removed from the test suite because they may have become obsolete or redundant after the addition. [15] Redundant test cases satisfy same requirements as other tests. Because the software is constantly retested, test suite sizes and running costs should be kept in control. [16] Retesting all tests that have ever been created for an application can be impossible in reality, which is why it has been stated that creating a small scale and near optimal test suite is a hot topic in software testing [17].

Test suite optimization can be referred as regression test suite optimization (RTO) when optimizing a regression suite. RTO is done to save time and resources but it is also possible to identify other objective functions for test suite optimization. They vary a little bit depending on if one is doing white-box testing or black-box testing. In Figure 1, we can see what objectives have been used previously. It has been stated that RTO is not safe if used with a single objective, because it will ignore vital test cases. Therefore it has to have multiple objectives. [18]

Objective	White-box based RTO	Black-box based RTO
Increase defect/faults detection rate	X	X
Increase defect detection capability	X	
Increase cost effectiveness	X	
Increase code coverage	X	
Increase function coverage	X	
Increase statement coverage	X	
Decrease execution time	X	X
Implementation complexity	X	X
Changes in class diagram	X	
Decision trees	X	
Decrease cost	X	X
Requirement change		X
Test case behavior in previous release		X
Architecture changes		X
Configuration files changes		X
Database files changes		X
User session		X
Customer priority		X
Requirement traceability		X
Fault impact of requirement		X

Figure 1. Black-box and white-box based RTO objectives [18]

Mathematically, RTO problem is NP-complete [18] which means that it is not known to have a polynomial time or in other words efficient algorithm [19]. But this does not mean that it could not have. In fact, if an efficient algorithm exists for an NP-complete problem, efficient algorithms would exist for all of them. [12 p.9-10] The models of solving RTO problem are selection, prioritization and minimization [18].

The terms used for test suite optimization are in my opinion a little bit confusing because there are many names used for test suite optimization in literature: RTO [18], RTS [2] and test suite reduction [20]. While the definition of these may differ to some extent, from now on, things are kept simple and we only talk about test suite optimization. Reduced test suite is the output of the test suite optimization, similarly to (Jeffrey, D. et al. 2005) [20].

2.2. Test suite optimization

2.2.1. Heuristic methods

Heuristic methods can be used to approximate the optimal solution of test suite optimization. [15]

Heuristic methods of test suite optimization can be divided to minimization and prioritization [18]. Test suite minimization can be defined as finding a minimal subset of test cases so that certain required coverage can be still achieved. On the other hand in test suite prioritization, test cases are sorted based on their cost per additional coverage so that the ones with the cheapest additional coverage are on the top of the list. After the sorting, a certain constant number of the top cases can be selected for the reduced test suite. [21] The other way to look at prioritizing is that it can provide quicker fault detection if test cases with coverage on modified and affected area of code are run first [22].

There are variations on minimization and prioritization in practice. We can use different criteria for both. [18] For example, one could use a technique where given that one knows the modified and deleted lines of a modified source code as well as the corresponding line numbers and the test case line coverage history (lines that each test case executes), one can prioritize test cases so that they execute at least once the modified lines and the lines after the deleted lines [4].

However, according to a recent study, code coverage may not be strongly connected to test suite effectiveness with only a low to moderate correlation between these two when the size of the test suite is controlled. The strength of this correlation has been found to vary between software systems, which is one of the major reasons why the strong correlation cannot be assumed. This can be said for most code coverage types. That said, code coverage is a good tool to find the under tested parts of the software but possibly not the overly tested. [23]

Similarly to coverage, software requirements can be used in test suite minimization. Given that we have a test suite T of test cases $\{t_1, t_2, \dots, t_n\}$ and requirements $R = \{r_1, r_2, \dots, r_m\}$, we must cover the same requirements as T with a minimal subset T' of T. However, because this is a NP complete problem, we need to use heuristics. [24]

A simple solution is a greedy heuristic algorithm which is based on classical approximation. It is run in iterations until all requirements have been covered. In each iteration, it selects a test case that satisfies the most requirements and throws away the satisfied requirements. In case of a tie, the test case is selected randomly among the tied test cases. The problem is that this method will lead to some redundant test cases. [24] We can improve the solution with HGS (Harold, Gupta and Soffa) algorithm. In HGS, a test case subset T_i of T consists of test cases that satisfy requirement r_i. Thus, the reduced test suite must contain one test case from each T_i in order to satisfy all R. The algorithm begins by considering subsets $T_i$of cardinality one $C = | T_i | = 1$, in other words, subsets that consists of one test case. These test cases are included in the reduced test suite and the T_i:s that contain these tests are marked. Next, T_is of cardinality two to the maximum are considered sequentially. In a particular cardinality C iteration, a test case with maximum number of occurrences in unmarked T_is is chosen for each unmarked T_i of cardinality C. For tie braking, further cardinality occurrences are considered. This solution is not perfect either because it also selects some redundant test cases. [15] Additional improvements can be made by, for example, introducing context tables and reducing the sizes of them by applying object and attribute reductions [24].

Due to test suite minimization leading to possible compromises in fault detection, there have been studies on test suite minimization with selective redundancy [20, 25]. One of the ways this could be done is by modifying existing heuristics such as HGS and creating a secondary criterion for selecting test cases. Definition use pair coverage can be used for this secondary criterion, meaning that test cases which contributed to this kind of coverage are also added to the reduced test suite. [20] Another way of keeping more redundant test cases in reduced test suite is to use multiple criteria. If a test case is selected for the reduced test suite with certain criterion and some other cases are other cases are redundant based on that criterion, other criteria are considered. If those other cases satisfy additional requirements with some other criterion, they are added to the reduced test suite. [25]

It has been pointed out that non-code changes should affect test suite minimization as well as the code changes. Examples of non-code changes would be changes to configuration files and databases. These components impact on the behavior of the software. As a result, affected test cases should be retested in case of these changes. By first gathering the needed information from test cases and their traces, test suite minimization can be done with heuristic methods. [26]

In the field of conservative regression testing, the inputs and region of a failure are recorded as well as the necessary steps to repeat it. By using this information, it is possible use a constraint-based test case minimization method. A test constraint, in this case, describes the ways how a certain failure can be detected. Moving away from requirements and code coverage, one is able to use these constrains for heuristics in order to select test cases for the reduced test suite by searching for a subset of test cases that detect all known failures. This can be done by constructing a constraint hierarchy using the steps of the constraints. [27]

Another hierarchical heuristic method is claimed to be able to minimize a suite with even million test cases in adequate time. The method starts by creating a mapping of test cases to functions by function coverage of each test case. Afterwards, it partitions test cases with similar set of function calls in same clusters. In practice, the criterion for clustering can be a threshold of mismatching function calls. In following steps, test cases are compared by their functional flow, line coverage and branch coverage one cluster at a time. [3]

2.2.2. Computational intelligence methods

The claimed successor of artificial intelligence, computational intelligence is inspired by biology and nature [18, 28]. Computational intelligence can be used to solve very difficult problems that may be considerably nonlinear, discontinuous, correlated or complex [28]. With test suite optimization computation intelligence can be identified to appear in form of genetic algorithms, swarm algorithms and fuzzy logic [18].

Genetic algorithms belong to a higher class called evolutionary computation. Evolutionary computation involves natural evolution where “survival of the fittest” applies. [28] Genetic algorithms can be divided into three phases: initializing a population of randomly generated individuals, evaluating individuals and selecting individuals which are combined and mutated to generate new individuals. The last two phases are repeated until some condition is met. The individuals represent a possible solution. [29] From the objective function point of view, test suite optimization with genetic algorithms can rely on code coverage or run time among others. This information is used on the fitness value when evaluating individuals for next generation. [18] Probabilities for each individual to be selected for next generation are computed from the fitness values. One way to do this is to use the strategy of roulette wheel where probability is in direct proportion to the fitness. [30]

Swarm algorithms are the methods of swarm intelligence. These algorithms are based on decentralized co-operation of simple individuals which are usually unintelligent. Swarm intelligence has been researched with inspirations from biological observations of, for example, ant colonies. The pros of swarm algorithms are good fault tolerance and simplicity of the individuals which may simplify the development process as well. [31] The individuals in a swarm can be considered as particles. Hence, we can talk about particle swarm optimization (PSO). In PSO, a single particle represents a candidate solution to the optimization problem. The particle flies in the search space according to its current velocity which is based on its memory and the memory of its neighbors. The search space represents all the possible solutions. [32]

PSO can be used in multi-objective test suite optimization [32]. Let's look at an example algorithm. First, a swarm with particles of random velocity and position is initialized. Second, the particles are evaluated according to the objective functions and the best particles are selected. Third, particle velocity and positions are updated according to the best particle in the population. Fourth, the population is crossed over two by two to create off-spring. Fifth, new off-spring is evaluated and the best one is selected. If the stopping criteria are met, the selected particle will be the final solution. If not, the algorithm continues from the third step. One can see the similarities of this algorithm to genetic algorithms. In fact, this is called a hybrid prioritized algorithm since it uses both PSO and a genetic algorithm. [33]

Moving on to fuzzy logic, there are a couple of capabilities it provides which can be seen as attempts at formalizing human decision making and reasoning. First, humans make decisions in situations where the information can be uncertain, imprecise, incomplete and conflicted. Secondly, these decisions can be done rationally without any measurements or computations. [34] Unlike bivalent logic and crisp set theory of zeroes and ones, in fuzzy logic, fuzzy sets are characterized with continuous range from zero to one [35]. Thus, fuzzy logic maps input vector into a scalar output. Fuzzy logic can handle simultaneously both numerical data and linguistic knowledge. [36]

In the context of test suite optimization, fuzzy logic can be used for example to make a decision in a tradeoff triangle. In a tradeoff triangle by going towards one corner (representing an objective) of the triangle, one goes farther from the others. Due to this being a problem in continuous domain, fuzzy logic suites it well. The idea is to first compute the membership value for test cases by each objective, which could be for instance performance, throughput and code coverage. The used fuzzy sets can be named as not suited, suited and best suited. After this one can determine the final crisp suitability of each test case with a method in which the center of 'mass' is calculated. [37]

2.2.3. Trace analysis methods

Test suite optimization based on URL traces has been demonstrated to be successful in an environment where user sessions have been traced to generate test cases automatically. In that environment, each user session starts when a new user, who is detected by a new IP-address, makes an HTTP request for an URL. The session ends when the user either leaves the website or the session times out. The HTTP requests for the URLs are logged for each user session. Thus, a test case consists of a sequence of HTTP requests i.e. a sequence of URL traces. The number of sessions is large and contains a vast amount of redundant traces. The similarity of two test cases is measured by the length of the longest common prefix of their traces [30]

First, a user session is redundant and can be removed if its trace is a prefix of another sessions' trace because the functionality of the first one is tested in second one. This is said to reduce the number of user sessions greatly while meeting the same requirements as the original complete set of user sessions. [30]

Second, elementary programming can be applied to group and prioritize the reduced set of user sessions. By grouping user sessions based on their similarity several test cases can be eventually run in parallel. The sessions with the length of the longest common prefix in the same predefined threshold value range are grouped together, forming four test suites in the study. To prioritize the test cases, the ones with the least similarity i.e. suite with the lowest threshold range are prioritized to be executed first. Inside the suite, test cases with the longest URL trace are prioritized first. [30]

Third however, a genetic algorithm can be applied to improve the prioritization because elementary prioritizing does not provide a very good approximation of the best solution. It is suggested that fitness values are based on the length of the longest common prefixes where a short longest common prefix results in a good fitness value. In the cross-over phase new user sessions can be created which include requests from various user sessions. [30]

2.3. Detecting redundancy

2.3.1. Overview

It can be easily seen in chapter 2.2 that the methods of test suite optimization require information of what test cases share same requirements, coverage or something else with each other. In other words, they require the information of what is redundant in order to select the cheapest test cases. This information is not available straight away. Thus, this chapter considers the different ways of telling how different and similar a sequence is from the others.

In the case study, test traces are strings but to simplify our problem for this literature review, let’s consider that each trace in a test case can be represented as a letter instead of a full string.

2.3.2. Exact string matching

Exact string matching problem is about finding all occurrences of a pattern i.e. string in larger amount of text [12 p.985]. To divide it into more specific problems, let’s consider the fact that when the strings are known. First, both the pattern and the text are known at the same time. Second, the text is known for some time and then some patterns are searched from it. Third and a reverse of second, the pattern is known for some time and some texts are then searched with the pattern. [38 p.122-123] In a typical situation, the problem at hand is the second one: there is a text available where a reader wants to find the occurrences of a pattern [12 p.985]. Determining which string is kept fixed is important because that string can be preprocessed to make the matching more efficient [38 p.122-123].

In worst case, naïve brute-force algorithm for the second case has matching time complexity $O((n-m+1)m)$ without any preprocessing. But currently using Knuth-Morris-Pratt algorithm, both the preprocessing and matching can be done in linear time $O(m)$ and $O(n)$ respectively. [12 p.986]

2.3.3. Longest common substring problem

Longest common substring problem should not be confused with the longest common subsequence problem which is presented in 2.3.4. The problems are different. [38 p.125] Longest common substring can be abbreviated to LCS [39] but for clarity, LCS is used only for longest common subsequence in this thesis.

Suppose we have a set of strings $S = \{S_{1,}, \ldots, S_n\}$ and an integer $2 \leq K \leq n$. In longest common substring problem the goal is to find the longest string that is a substring of at least K strings in S. [39]

The problem can be solved using a suffix array which will be presented later in chapter 3.1.3. The key point in the solution is the fact that a longest common substring for two strings is the maximum longest common prefix of their suffixes. [39]

2.3.4. Longest common subsequence problem

Longest common subsequence (LCS) is used often in DNA comparisons. A DNA strand consists of strings of molecules. There are four possible molecules which can be represented with a capital letter in comparisons: adenine (A), guanine (G), cytosine (C) and thymine (T). One objective in the comparison is to find out how similar two or more strings are. In LCS, this is expressed by finding a common string with maximum length that appears in all the compared strings in the same order but not necessarily consecutively. Thus, a subsequence of a given sequence is the given sequence where some elements may or may not have been removed. [12 p.390-391]

Let's look at an example. Assume, we have two strings $S_1 = 'AGCTAC'$ and $S_2 = 'CACGTA'$. For instance $S_3 = 'GTA'$ with a length of three is a common subsequence (CS) to both S_1 and S_2 but not a longest one. Instead, a LCS would be $S_4 = 'ACTA'$ with a length of four. But it would not be the only one because $S_5 = 'AGTA'$ would be also length four of and a LCS.

To calculate LCS for S_1 and S_2 by a brute-force algorithm, one would have to first enumerate all subsequences of S_1. Secondly, each subsequnces would have to be checked if it is a subsequence of S_2 while keeping the currently longest common subsequence in memory. If S_1 has n elements it can be proven to have 2^n subsequences. Because of this, brute-force algorithm has exponential time complexity. [12 p.392]

2.3.5. Dynamic programming

Review of dynamic programming is included in this thesis because it is usually applied to optimization problems and can be used to solve the LCS problem. Dynamic programming can be used to solve problems by dividing them into multiple problems, subproblems. There may be some overlapping in the subproblems. However, all subproblems are solved only once even if they occur multiple times. This is done by saving the solution of each subproblem to a table after the subproblem is solved. To generalize, dynamic programming can be divided into four steps: [12 p.359-360]

1. Characterization of the structure of an optimal solution.
2. Recursively defining the value of an optimal solution

3. Computation of the value of an optimal solution, generally from bottom to up
4. Construction of an optimal solution from the computed data

Step 4 can be skipped if only an optimal value is needed and not the solution. When also step 4 is included, one usually will have to retain some additional data in step 3 to compute the corresponding solution efficiently. [12 p.359]

Dynamic programming may be applied when the problem at hand exhibits optimal substructure and consists of many overlapping subproblems. Time- and resource-wise, many overlapping subproblems should be obvious here because it will enable the algorithm to run faster and not compute solutions to new subproblems time and time again. On the other hand, optimal substructure in a problem means that an optimal solution contains the optimal solutions to its subproblems. This is valuable in dynamic programming because an optimal solution to a problem is built from the optimal solution of the subproblems. [12 p.379-384]

To solve LCS problem with dynamic programming, a score matrix [40] $c[i,j]$, which contains the lengths of LCS of S_1 and S_2, must be calculated with (2.1) [12 p.390-393]. In addition to the matched positions, the score matrix also contains the unmatched positions. By matched positions, we mean the occurrences of LCS. [40]

$$c[i,j] = \begin{cases} 0 & if\ i = 0 \lor j = 0 \\ c[i-1,j-1]+1 & if\ i \neq 0 \neq j \land S_1[i] = S_2[j] \\ \max\left(c[i,j-1], L[i-1,j]\right. & if\ i \neq 0 \neq j \land S_1[i] \neq S_2[j] \end{cases} \tag{2.1}$$

In the formula, we can see that it calls itself recursively and the if-statements define which subproblems are covered [12 p. 393]. The formula can be improved. For example, it can be proven that the non-occurrences of LCS do not have to be calculated in order to save computation time. [40]

2.3.6. Sequence alignment methods

Sequence alignment methods can be divided to global and local alignment methods. In global alignment, the goal is to compare similarities of two or more sequences from end to end and. In local alignment, the aim is to compare most similar regions within the sequences. [41] Another division can be made on the number of sequences on the alignment. In the simplest case, pair-wise sequence alignment considers two sequences at a time while multiple sequence alignment (MSA) considers three or more at a time. [42] Like LCS, sequence alignment has been proven useful in DNA and protein sequence analysis [41].

The optimization problem in sequence alignment algorithms is to maximize the number of positions of matching characters in the strings while minimizing number of mismatches. This can be done by adding gaps (denoted by '-') in to the strings to move characters to the right for better matching. Alignment is assessed with a score function and penalties. The simplest way is to assign a score or penalty for three possible cases: matching character, mismatching character and gap. [42] Normalized LCS can also be used to calculate the score of sequence alignment by calculating the matching characters and dividing it by the length of the alignment [43]. However in biology, scorings are usually considerably more complicated because there are usually more specific cases where more than just one character has to be taken in account and the location of a difference is important [42]. On the other hand, a difference has to be noted between biology and normal texts. In biology, the motivation of sequence comparison arises from the fact that organisms are all related by evolution which leads to sequences having a lot of similarities. The same cannot be said for normal text [44].

Like with LCS, dynamic programming algorithms can be used to calculate pair-wise sequence alignment and produce the optimal alignment [44]. Let's look at Needleman and Wunsch's full-matrix algorithm. One sequence corresponds to the columns and the other to the rows. Cells in the matrix are computed left to right and top to bottom with the optimal score in the bottom-right corner. A path between the top-left and bottom-right corner represents an alignment. Moving horizontally or vertically represents inserting a gap in the opposing sequence. To compute optimal solutions, one has to start from the bottom-right corner and fall back to the top-left corner along an optimal path. [42]

Pair-wise alignment	**Matches**	**Mismatches**	**Gaps**	**Norm. LCS**
T A G A T A C G A	6	1	2	$\frac{6}{9}=\frac{2}{3}$
C A G - T A C - A				
T A G A - T A C G A	5	2	3	$\frac{5}{10}=\frac{1}{2}$
C - G A T T A - G C				
C - A G T A - C A	5	1	3	$\frac{5}{9}$
C G A T T A G C -				

Figure 2. Example pair-wise sequence alignments of strings TAGATACGA, CAGTACA and CGATTAGC

Computations in each cell are very light. Thus, the time complexity $O(n^2)$ does not really matter even with large sequences. However, the same space complexity does because the matrix gets very large quite fast. The algorithm can be applied also to MSA. In this case, the time and space complexity is $O(n^d)$ where d is the number of sequences. This is quite bad memory-wise. However the memory requirements have been reduced with, for example, Hirsberg's algorithm and FastLSA algorithm which do not store more than couple of rows at a time and do some recomputations of the cells instead. [42] Nevertheless, the amount of sequences is nowadays very large and there are problems when aligning many long sequences together. It can also be proven that MSA is an NP-complete problem as well. [45]

To go even more efficiently, there have been uses of heuristic and probabilistic methods. As we know, these are approximation methods and may not provide optimal but a sort of "best guess" alignment. Currently, one of the most popular heuristic MSA methods is progressive alignment. As the name suggests, the full alignment is built progressively by first computing pair-wise alignments. After that the sequences are clustered together by their similarity scores in order to see the relationships between them. From the similarity scores one can compute distance scores and afterwards a guide tree based on the distance scores. According to the guide tree, the sequences are then added one by one to the alignment and thus forming a MSA. [45]

3. METHODS

3.1. Relevant methods

The relevant methods to the case study are supermaximal repeats, suffix tree and suffix array. All of the methods are applied to a test trace consisting of a traced message sequence. We consider a repeat in a test trace as a sequence of traced messages which appears two or more times in the test trace. Supermaximal repeats are relevant because they were used as the starting point for the pair-wise supermaximal repeats which are presented in the selected methods. Similarly, suffix array acted as the solid building block for the selected enhanced suffix array method. Suffix tree is presented here for comparison to the suffix array method because of its importance on string processing [46].

3.1.1. Supermaximal repeats

Let us once again consider a string S and an example string $S_e = 'abceabcd'$. If and only if $(i_1, j_1) \neq (i_2, j_2) \land S[i_1, \dots, j_1] = S[i_2, \dots, j_2]$ for set $R = \{(i_1, j_1), (i_2, j_2)\}$, R is a repeated pair. Thus, in S_e an example repeated pair is: $\{(0,1), (4,5)\}$ because $(0,1) \neq (4,5) \land S_e[0,1] = S_e[4,5] = 'ab'$. Other repeats would be $'a'$, $'b'$, $'c'$, $'bc'$ and $'abc'$. [46]

Secondly, a repeated pair R is left maximal if $S_e[i_1 - 1] \neq S_e[i_2 - 1]$ and right maximal if $S[j_1 + 1] \neq S[j_2 + 1]$. It is left maximal and right maximal if the pair cannot be extended to either direction and still be a repeated pair. In S_e repeated pair $'bc'$ would be considered right maximal but not left maximal because $S_e[3] = 'e' \neq S_e[7] = 'd'$ and $S_e[0] = S_e[4] = 'a'$. A repeated pair is maximal if it is both left and right maximal. In S_e, string $'abc'$ would be maximal repeat because $S_e[3] = 'e' \neq S_e[7] = 'd'$ and $S_e[-1] = '' \neq S_e[3] = 'e'$. [46]

A maximal repeat that never occurs as a substring of any other maximal repeat is called a supermaximal repeat [46]. In S_e, the only supermaximal repeat would be the only maximal repeat $'abc'$. However, in $S_{e2} = 'abceabcdabcd'$ the only supermaximal repeat would be $'abcd'$.

3.1.2. Suffix tree

In string processing, suffix tree has been stated as one of the most important data structures. Suffix tree has advantages in scenarios where the sequences to be analyzed are very large and stationary. Suffix trees have been used a lot in the field of bioinformatics where whole genomes are studied. Genomes, as we know, are very large.

Suffix tree could have become popular due to its linear time and space complexity $O(n)$ in storage and computation. However, suffix tree suffers from large space consumption because, in the worst case scenario, the construction of it requires 20 bytes per input character. Another problem related to memory is that in many cases the memory references are not good in terms of locality which causes performance losses in cached processor architectures. [46]

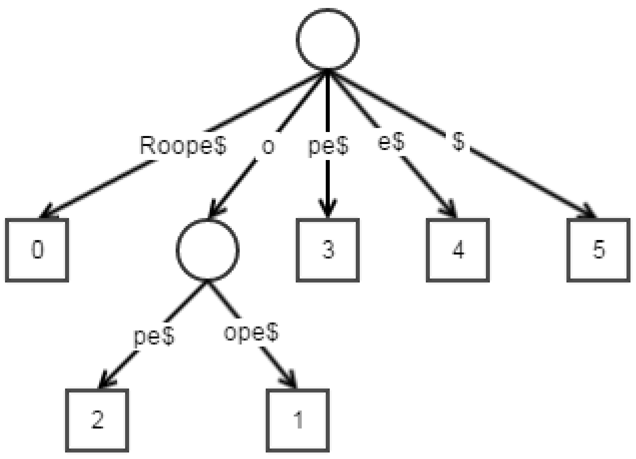


Figure 3. Suffix tree of string $S = 'Roope\$'$

Suffix tree is a rooted directed tree. It has exactly $n + 1$ leaves that are numbered from 0 to n, where n is the number of characters in the string. The numbering represents the start position of a suffix. The extra character at the end of the string is a terminal character \$ which is considered to be greater than any other character in the string. Concatenating edge-labels from root to a leaf forms a suffix of the string. And for example, to answer a question: "Is string $'ope'$ a substring of string $S = 'Roope\$'$?" walk the tree in the order of the characters in the word $'ope'$ starting from the root. If all letters are found, the answer is yes. In this case, you should end up in leaf labeled with 2 and yes is the correct answer. [46]

Moreover, following rules apply to suffix trees [46]:

- Every internal node has at least two children, except for the root
- Each edge has to be labeled with a non-empty substring of the string
- No edges beginning from a node can have same beginning characters

To find the supermaximal repeats from suffix tree, one must consider a couple of things. Let's define a left character as the left most character in an edge label to a node. Let's also define a left diverse node v as a node that has in its subtree at least two leaves with distinct left characters. Now, a supermaximal repeat is a left diverse node whose children are all leaves and each child has a distinct left character. [47] In the example string $S = '\text{Roope\$}'$, it is easy to see that string $'\text{o}'$ is a supermaximal repeat.

3.1.3. Suffix array

Suffix array requires less bytes per input characters than suffix tree, $4n$ to be exact [46]. The currently fastest parallel suffix tree construction algorithm requires the construction of suffix array and LCP array which is defined later in this chapter [50]. The drawback of the simplest and earliest suffix arrays was that in the worst case scenario a suffix array could be constructed in $O(n \log n)$. However this was a special case where the alphabet was small [48] and has since been improved to be able to be constructed in $O(n)$. Also the queries to suffix array seemed to suffer from similar problems and took $O(m + \log n)$ in the worst case, which has been improved to $O(n)$. [46]

Let us consider a string S. Suffix array A of S in its simplest form is a sorted list of all suffixes of S [48] and suffix table (suftab) which is an array of integers from 0 to n signifying the lexicographical order of the suffixes [49]. Lexicographical order in this case means that the suftab is ordered as the suffixes are ordered so that each integer is still on the same row as the corresponding suffix. [49]

But there has been development in the field of suffix arrays and recently enhanced suffix arrays have become very popular data structure for example biological data indexing. In enhanced suffix arrays, the basic suffix array is extended with various tables that vary depending on the problem at hand. [49] Very often suffix array are used with longest common prefix (LCP) information in LCP table (lcptab) [46] or LCP array [50]. This lcptab information is the LCP value between two adjacent elements in the suffix array [48].

To find supermaximal repeats with suffix array, one can enhance suffix array with Burrows and Wheeler transform. The output of Burrows and Wheeler transform (BWT) is saved into an array, bwttab, with the suffix array. Each $bwttab[i]$ for $suftab[i] \neq 0$ can be calculated in linear time: $bwttab[i] = S[suftab[i] - 1]$. If $suftab[i] = 0$, $bwttab[i]$ is undefined. [46]

The above tables were the ones that made it into the implementation in the case study but there are many more tables that can be used to enhance suffix array [46]. Instead of LCP array one can use permuted longest common prefix array (PCLP array) which is faster and more space efficient to construct. In PLCP array, the values are stored in the positional order instead of lexicographical order. PCLP array construction is a linear time algorithm. [51] However, in most applications PCLP array is not usable in itself and the LCP array has to be constructed [52]. This is not a problem because even combining LCP array construction from PLCP array and construction of the actual PCLP array itself became the fastest known algorithm for calculating the LCP array [51]. This was still the case at least in 2011 [52]. Also, by using this method it is possible to make a different tradeoff between time and space complexity [51]. The algorithm can be parallelized. Thus, one can expect 14.4–21.8 percent time savings in 40 core systems. [50]

i	suftab	lcptab	plcptab	bwttab	$S_{suftab[i]}$
0	4		1	d	aabceabcd$
1	0	1	0		abcdaabceabcd$
2	9	4	0	e	abcd$
3	5	3	0	a	abceabcd$
4	1	0		a	bcdaabceabcd$
5	10	3	3	a	bcd$
6	6	2	2	a	bceabcd$
7	2	0	0	b	cdaabceabcd$
8	11	2	0	b	cd$
9	7	0	4	b	ceabcd$
10	3	0	3	c	daabceabcd$
11	12	1	2	c	d$
12	8	0	1	c	eabcd$
13	13	0	0	d	$

Figure 4. Enhanced suffix array of word abcdaabceabcd$

3.2. Selected methods

The selected methods are full sequence plot, supermaximal repeats via enhanced suffix array, filtering detected repeats and test driven development. These methods were chosen so that, first of all, we can get an overview of the sequences in the full sequence plot and, secondly, get good amount of meaningful repeats to inspect with supermaximal repeats. Enhanced suffix array seems to be the most efficient way to find these repeats. Filtering repeats helps removing uninteresting repeats from the results while browsing them. On the other hand, Test driven development provides unit tests and allows continuous integration for the developed application.

3.2.1. Full sequence plot

To produce a plot of a full test trace, we assign a number for each traced message and plot them in the same plot one at time. The assigned value is explained in the case study. For the x-axis we use the position of each message in the sequence and for the y-axis the assigned value of the traced message. In order to see where test cases begin we plot straight lines to separate test cases from each other. We then try to find repeating patterns from the plot.

3.2.2. Supermaximal repeats via enhanced suffix array

An LCP-interval is an interval $[i, \dots j]$ of LCP value of ℓ if [46]:

$$lcptab[i] < \ell \tag{3.1}$$
$$lcptab[k] \geq \ell \;\forall\; k \in [i+1,\ j] \tag{3.2}$$
$$lcptab[k] = \ell \;\exists\; k \in [i+1,\ j] \tag{3.3}$$
$$lcptab[j+1] < \ell \tag{3.4}$$

An LCP-interval is a local maximum if [46]:

$$lcptab[k] = \ell \;\forall\; k \in [i+1,\ j] \tag{3.5}$$

If we combine (3.2), (3.3) and (3.5), we can clearly see that it is sufficient for $[i, \dots, j]$ to be a local maximum in the lcp-table if for any ℓ:

$$lcptab[i] < \ell \tag{3.6}$$
$$lcptab[k] = \ell \;\forall\; k \in [i+1,\ j] \tag{3.7}$$
$$lcptab[j+1] < \ell \tag{3.8}$$

It can be proven that an ℓ-interval and a local maximum $[i, \dots, j]$ is a supermaximal repeat if all elements in $\{\mathrm{bwttab[i]}, \mathrm{bwttab[i+1]}, \dots, \mathrm{bwttab[j]}\}$ are unique. The correct string to report as supermaximal repeat is $\omega = S[suftab[i], \dots, suftab[i] + \ell - 1]$. [46]

Let's introduce a new concept, pair-wise supermaximal repeats. If we have a string that can be divided into substrings with some knowledge of that string, we can look for supermaximal repeats in each of the combination of two substrings. To find further information about the similarities of the substrings, let's put a distinct character $'\$'$ between two substrings during the supermaximal repeat search. Formally, let's define pair-wise supermaximal repeats as supermaximal repeats of each concatenated two's combination of some substrings $\{S_0, \dots, S_m\}$ of a string S where the concatenation of the substrings forms the string S. The amount of combinations of two is $\binom{m}{2} = \frac{m!}{2!(m-2)!} = \frac{m!}{2(m-2)!}$.

For example, consider a string $S = 'abcdsabcdedabcs'$. The supermaximal repeats of this string would be $'abcd'$ and $'s'$. Assume, we know that $S_0 = S[0, \dots, 4] = 'abcds'$, $S_1 = S[5, \dots, 10] = 'abcded'$ and $S_2 = S[11, \dots, 14] = 'abcs'$. Then, there will be $\binom{3}{2} = 3$ combinations of two. The combinations of S_0, S_1 and S_2 would be $P_0 = 'abcds\$abcded'$, $P_1 = 'abcds\$abcs'$ and $P_2 = 'abcded\$abcs'$. Then, the supermaximal repeats that we would find are $R_0 = \{'abcd'\}$, for $R_1 = \{'abc', 's'\}$ and $R_2 = \{'abc', 'd'\}$ for P_0, P_1 and P_2 respectively. Thus, the pair-wise supermaximal repeats of S would be $\{'abc', 'abcd', 'd', 's'\}$.

In the above example, we can see that use of pair-wise supermaximal repeats instead of plain supermaximal repeats may lead to reports of similarities where they would not have been reported otherwise while leaving out repeats that always occur inside another repeat.

In our case study, test trace messages consist of strings instead of single character. Thus, when we apply a suffix array, we consider finding supermaximal repeats from a set of strings instead of a string. This will lead to the fact that a suffix and a prefix of the set contain set of strings as well, instead of a string. Thus, LCP value is calculated by the longest common prefix of sets, in other words, most common strings counting from the beginning. Similarly, BWT value will contain a string instead of a single character. With pair-wise supermaximal repeats, there is a distinct string instead of a distinct character between two set of strings. If we could assign a single character for a test trace message, we could ignore these changes and run suffix array normally. However, due to the unknown number of distinct messages, this is not possible. Case study environment is discussed more specifically in chapter 4.1. The found pair-wise supermaximals repeats are compared in overview to found supermaximal repeats in chapter 5.

3.2.3. Filtering detected repeats

Each repeat contains information about its length and how many times it occurs. These values are used as maximum and minimum filters. Each repeat also contains information of the sequence itself which is also used as a filter. In the case study, all filters can be adjusted manually while browsing the results. A few results with different filter settings are presented for the case study.

3.2.4. Test driven development

Most of the code for the case study of this thesis was written with Test driven development (TDD) method. TDD follows a pattern where tests are written before the production code. First step in development is to write a failing unit test. Second step is to run all tests and see the test fail. Third step is to write only the amount of production code that will make the failing test pass. No new production code should be written if all unit tests pass. Similarly, no new unit tests should be written if a unit test fails. Fourth step is to run the tests and see if all tests pass. Fifth step is to refactor the code to remove duplicates etc. Both the unit tests and the production code are refactored. After fifth step, the cycle is repeated starting from the first step: writing a failing unit test. [53 p.1-14]

Mocks are objects that act as other objects by imitating their behavior and therefore require Object-oriented programming (OOP). Mocks are often used in TDD because sometimes the unit tests would take too long to run or might run inconsistently otherwise. It is also possible that without mocking the tests may not be run at all and the test scenario would be very hard to recreate because of long dependency chains etc. With mocks, developer can test the interaction of an object to its neighboring objects by tracking what parameters are used to call a method and how many times certain method was called. In result, mocks make unit tests faster and easier. [54]

Code coverage and running times of the unit tests are presented in the results.

4. EXPERIMENTS

4.1. Case study CI environment

4.1.1. Jenkins

Jenkins is an open source continuous integration server that can be used to run and monitor repeated tasks automatically. These tasks include building and testing software, but virtually, Jenkins can run any given task which can be automated. Whenever a build or test fails an email or RSS can be sent. Because Jenkins can distribute build loads to multiple computers, Jenkins is also a good tool for monitoring the executions of externally run jobs like cron jobs. Jenkins is used in various companies such as Atmel, Dell, eBay, Facebook, GitHub, NASA, Netflix, Sony, Tumblr and Yahoo!. Total number of reported Jenkins installations is over 100 000. There are also many Jenkins servers publicly available and Jenkins is used in some open source projects. [55]

Jenkins can be extended by using the various plugins that already exist or by writing a new one. There are many plugins for source code management, build triggers, build tools, build wrappers, build notifiers, slave launchers and controllers, build reports, publishing, other post-build actions, UI, authentication and user management, parameters and many other things. [55]

Every task in Jenkins is run in a job. Each execution of a job is referred to as a build. Builds can have different outcomes. A few of the most common are successful, unstable and failed. When a build is successful, it was built with no errors. On the other hand, if some errors occur, for example failed test cases or some other error is reported, the build is usually unstable. However, build is considered as failed if during the build something broke and for example needed output files could not be produced. There are also configuration options to set builds failed if for instance some percentage of test cases failed. [55]

Jobs can launch other jobs depending on their status. For example, application can be continuously published when certain jobs are successful. A job, which is launched by other jobs, is called their downstream job. And in reverse, a job that launches other jobs is called their upstream job. [55]

4.1.2. Robot Framework

Robot Framework is an open source python-based tool designed for test automation. Nokia Networks supports the development of the core framework. [56] Robot Framework can be extended on top of the built-in keywords. The key uses lie in the field of acceptance testing. Robot Framework enables testing distributed and heterogeneous applications that have to be verified using various interfaces and technologies. This is possible because the core framework knows nothing about the target system that is tested and the test libraries handle the interaction as can be seen from the high-level architecture in Figure 5. Keywords in Robot Framework are basically functions and can both accept arguments as well as return values. Each high-level keyword consists of lower level keywords. Ultimately, a lowest level keyword executes a simple action like transmitting a message. [9] In this thesis, we will refer to the path from the highest to lowest level keyword as keyword path.

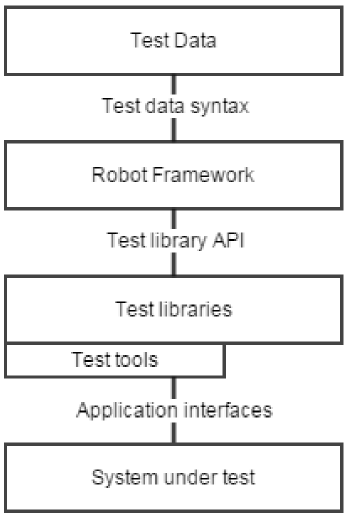


Figure 5. Robot Framework high-level architecture

Test suites in Robot Framework are directories and files. Because of the tabular syntax and two spaces acting as a separator, test cases can be written to be read easily because keywords and names allow one space between words. Keywords and test cases can also be written in Python and in Java when running Robot Framework on Jython. Writing libraries in C is also supported by the Python C API or through Python's ctypes module. [9]

Robot Framework also supports creating test templates which convert the common keyword-driven test practice into data-driven tests. This allows creating test cases with same keyword calls without specifying the keywords explicitly in each test case with the same keyword pattern. In these "templated" test cases, only the used template test case and arguments to the keywords are specified. [56]

In addition to test suites, tests to be run can be chosen by different parameters. Simple way is to specify test names by using *--test* command line parameter but test cases can be also tagged. This way it is possible to include test cases that contain a certain tag and exclude test cases that contain another. Among these there are many other test case selector parameters. One can include non-critical tests or critical tests. Other possibility is to use .txt argument files to define test cases that should be run. [9]

Robot Framework provides HTML reports and XML output files for the tests [9]. There is a Jenkins plugin for Robot Framework that can be used for collecting and publishing the results in Jenkins [56]. The outputs can be post-processed using Rebot tool [9].

There are five log levels in Robot Framework. In the ascending order of the amount of detail, they are FAIL, WARN, INFO, DEBUG and TRACE. TRACE is the most detailed level that logs even the keyword arguments, timestamps and return values while FAIL can only be used by the Robot Framework itself. [9]

4.1.3. Case study software component testing

There are multiple CI systems in use for Nokia Networks but for this case study, we are concentrating on only one of them and particularly on system component testing part of it. For the remaining part of this thesis, let's refer to this area as case study CI. Let's also define that the software application that this case study CI tests is called case study software application and the testing itself is called case study SCT.

In the case study CI, the main tools for the running CI are Jenkins, Robot Framework, Subversion (SVN), test servers, test PCs, cloud computers, test units where the application being tested is run, Jenkins radiator PCs and reservation system for test units. The developed case study software application along with case study SCT tests lie in SVN version control repository where the code can be checkout to test servers. These test servers are used by the developers and Jenkins which runs tests automatically. The test servers mainly exist for developer and Jenkins workspaces.

The test units can be reserved automatically when tests are run regardless of whether the tester is a developer or Jenkins. Test PCs can be either virtual or physical but they are connected to the test units and send messages between each other which can be traced and will be referred to as case study test trace messages for the remainder of the thesis.

Cloud computers can be used for compiling the case study software application and for other tasks which do not require the test units. Jenkins radiator PCs show the current status of the Jenkins builds to developers.

There are different regression tests run in Jenkins products. For the case study, we choose one of the most actively run regression test job which takes for average 20.9 minutes to run while the standard deviation is 5.29 minutes. These numbers were calculated from the last 247 builds. At the time of writing, the number of test cases in the regression test is 645. The regression test is divided to 12 sections which run in parallel. This means that 12 test cases can be in execution at a given time, dividing the total execution time by 12 in theory. However in practice, sections tend to have different durations, initial test environment setups have to be done for each section and the results of each section have to be joined together. These all add to total execution time.

The tests are run with Robot Framework with various custom Python and Robot keywords to test different functionalities of the software. Basically, it setups the test unit and gives actions to the test unit by sending messages and asserts the responses. Most teardowns are also done by sending messages. Because of this, most of the test sequence information is in the traced messages. The messages can be seen as an application interface in Figure 5.

The messages in the test trace consist of a header and the rest of the message. In the header, we can find two important parts for our case study. We have an ID (msgId) and a corresponding type name for the message. Rest of the header information is not relevant to the case study.

Test case durations and outcomes are saved in a database for longer period of time than in Jenkins. From there one is able to look for test case execution history, for example, when the test case has failed before etc.

4.1.4. Command line tool of MRS Finder

The developed application in this thesis, MRS Finder (Maximal Repeat Sequence Finder), consists of two parts: Python command line tool and HTML results browser. We are not going to discuss the whole implementation because that would take quite a lot of pages. Instead, let's consider what was implemented as well as what issues that arose and how they were handled during the development. We start by looking at the command line tool.

Let's begin with the input and output specifications which are described in Figure 6. For input, we take in the *output.xml* that is generated by the Robot Framework after tests have been run [9]. Other inputs are the command line parameters that can be used to get different outputs. Main outputs are two JSON files. *msgs.json* file which contains the messages in smaller format than the output.xml and *seqs.json* file which contains the repeats that were found. Other output variant is a plot with the msgId:s in decimal format on the y-axel and message index number on the x-axel. In the plot, a line from the bottom to top separates test cases visually. Later on, we will refer to this plot as msgId plot.

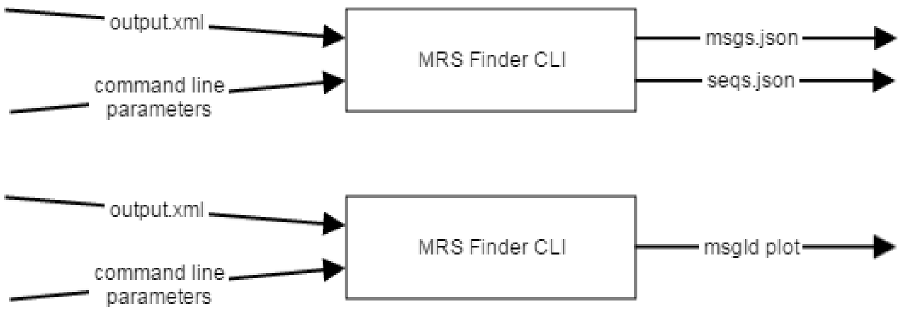


Figure 6. Input and output variants of command-line interface (CLI). Different command line parameters yield either JSON files or msgId plot.

Command line tool was developed with test driven development (TDD). In this project, unittests were written using Python's "de facto" standard unit testing framework, *unittest* library [57]. A Jenkins job was also configured to continuously run the unit tests while commits to the repository of the application were made.

Moving on to XML-parsing, there was already code available for printing all the traced messages and the corresponding keyword path from the Robot Framework's *output.xml*. However, we needed to store them instead of printing to construct suffix arrays. To store long messages more memory efficiently, we hash the message after the header with MD5 hash function. This leads to a message containing a header and a checksum of the remainder of the message. We store only the messages sent to the test unit because two sequences with same sent messages should have also same received messages application-wise in theory. Robot Framework keyword timestamps, msgId:s and test case names were also stored for each message. Test case start indexes were also stored for easier combination creation when searching for pair-wise supermaximal repeats. If a keyword failed in a test or message cannot be parsed from the trace, it is skipped.

Computation of maximal, supermaximal repeats and pair-wise supermaximal repeats are described in chapter 3.2.2. The only addition that should be mentioned here is that sorting the list of suffixes was done without any distinct character or string at the end of the suffix. Instead, if two lists were the same until the end but either was shorter, the shorter list would be considered greater and therefore act as it had a distinct greatest character or string at the end. Sorting the suffix array is done by using Python's built-in *sort()* method for lists where an item in the list is a suffix and a suffix contains messages i.e. strings.

Like mentioned in chapter 4.1.3, a regression test build contains 645 test cases, which means that for pair-wise supermaximal repeats we would have to consider $\binom{645}{2} = 207\,690$ combinations of two. However, we also have 12 section setups which raises our number of combinations of two to $\binom{657}{2} = 215\,496$. However, some cases execute teardowns outside the actual test case. Therefore, there are actually even more combinations of two, $218\,791$ to be exact. To counter the amount of suffix arrays, the supermaximal repeats for the each combination can be computed in parallel threads. The number of threads is configurable in the input parameters. The workload is divided to threads so that each thread gets same amount of pseudo-randomly selected combinations to calculate. Duplicate pair-wise supermaximal repeats that appear in one thread can be removed inside the thread. However, because we may get duplicate repeats between threads, we have to remove those in the main thread.

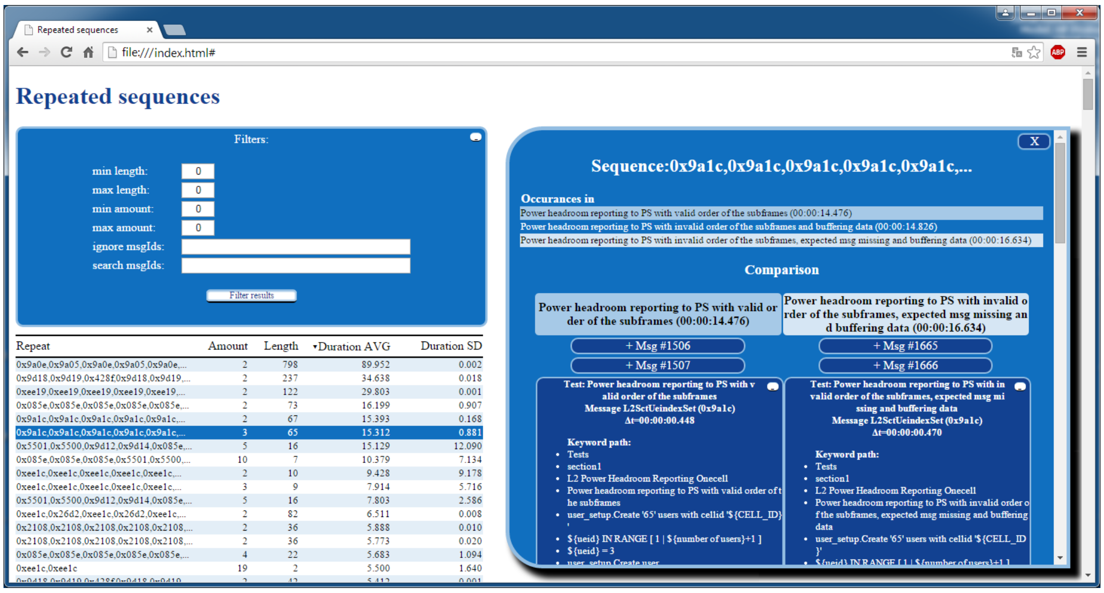
Python library	**Library description [58]**	**Reason of use**
datetime	Handles date and time operations	Converting Robot Framework timestamps to more familiar format.
glob	Unix style pathname pattern expansion	Finding files that match a pattern in unit tests
hashlib	Interface to many secure hash and message digest algorithms	MD5 hashing of messages while parsing output.xml
itertools	Functions creating iterators for efficient looping	Iterating over combinations
json	JSON API for Python	Creating test cases. Creating output in JSON format.
matplotlib	2D plotting library. pyplot module provides MATLAB-like interface [59]	Showing and interacting with msgId plots.
multiprocessing	Supports spawning processes to get around Global Interperter Lock	Parallel processing
os	Miscellaneous operating system interfaces	Enable writing and reading files in different directories.
random	Pseudo-random number generators	Pseudo-randomly dividing workload for threads
re	Regular expression operations	Finding traced messages in the output.xml
shutil	High-level file operations	Recursively deleting testing directory in unit test teardowns
subprocess	Allows spawning new processes	Unit testing command line parameters of the implementation.
sys	System-specific parameters and functions	Finding Python version in use
time	Various time-related functions	Measuring durations of functionalities
unittest	Python unit test framework	Creating and running unit tests
xml.etree.cElementTree	C implementation of XML API for Python.	Reading output.xml.

Figure 7. Python implementation dependencies

Although we are talking about threads, Python's *threading* library is not used but Python's *multiprocessing* library. This is because the *threading.Thread* object cannot utilize multiple processor cores because of CPython's global interpreter lock [58]. However, *multiprocessing.Process* object can utilize all cores because it uses subprocesses to run the threads [60]. When *multiprocessing.Process* object's *start* method is called it actually starts a subprocess [60]. All Python implementation dependencies are presented in Figure 7.

The command line tool is run on a cloud computer with 30 cores and 2 TB of RAM. In the case study CI, tests are not normally traced because the output files sizes are then considerably larger and the results should be kept for a while. When tests are run with traces, *output.xml* can be over 1.9 GB in file size. However, if tracing is not used and some passed keywords are removed from *output.xml*, the file size is around 30 MB. This is why traced tests are run only once a day by a different Jenkins job. The implementation itself is run in a downstream Jenkins job of the tracing job.

4.1.5. Results browser of MRS Finder



Repeated sequences

Filters:

min length:	0
max length:	0
min amount:	0
max amount:	0
ignore msgIds:	
search msgIds:	

Repeat	Amount	Length	▾Duration AVG	Duration SD
0x9a0e,0x9a05,0x9a0e,0x9a05,0x9a0e,...	2	798	89.952	0.002
0x9d18,0x9d19,0x428f,0x9d18,0x9d19,...	2	237	34.638	0.018
0xee19,0xee19,0xee19,0xee19,0xee19,...	2	122	29.803	0.001
0x085e,0x085e,0x085e,0x085e,0x085e,...	2	73	16.199	0.907
0x9a1c,0x9a1c,0x9a1c,0x9a1c,0x9a1c,...	2	67	15.393	0.168
0x9a1c,0x9a1c,0x9a1c,0x9a1c,0x9a1c,...	3	65	15.312	0.881
0x5501,0x5500,0x9d12,0x9d14,0x085e,...	5	16	15.129	12.090
0x085e,0x085e,0x085e,0x5501,0x5500,...	10	7	10.379	7.134
0xee1c,0xee1c,0xee1c,0xee1c,0xee1c,...	2	10	9.428	9.178
0xee1c,0xee1c,0xee1c,0xee1c,0xee1c,...	3	9	7.914	5.716
0x5501,0x5500,0x9d12,0x9d14,0x085e,...	5	16	7.803	2.586
0xee1c,0x26d2,0xee1c,0x26d2,0xee1c,...	2	82	6.511	0.008
0x2108,0x2108,0x2108,0x2108,0x2108,...	2	36	5.888	0.010
0x2108,0x2108,0x2108,0x2108,0x2108,...	2	36	5.773	0.020
0x085e,0x085e,0x085e,0x085e,0x085e,...	4	22	5.683	1.094
0xee1c,0xee1c	19	2	5.500	1.640

Sequence:0x9a1c,0x9a1c,0x9a1c,0x9a1c,0x9a1c,...

Occurances in

Power headroom reporting to PS with valid order of the subframes (00:00:14.476)
Power headroom reporting to PS with invalid order of the subframes and buffering data (00:00:14.826)
Power headroom reporting to PS with invalid order of the subframes, expected msg missing and buffering data (00:00:16.634)

Comparison

Power headroom reporting to PS with valid order of the subframes (00:00:14.476)	Power headroom reporting to PS with invalid order of the subframes, expected msg missing and buffering data (00:00:16.634)

Test: Power headroom reporting to PS with valid order of the subframes
Message L2SctUeindexSet (0x9a1c)
Δt=00:00:00.448

Keyword path:
- Tests
- section1
- L2 Power Headroom Reporting Onecell
- Power headroom reporting to PS with valid order of the subframes
- user_setup.Create '65' users with cellid '${CELL_ID}'
- ${ueid} IN RANGE [1 | ${number of users}+1]
- ${ueid} = 3
- user_setup.Create user

Test: Power headroom reporting to PS with invalid order of the subframes, expected msg missing and buffering data
Message L2SctUeindexSet (0x9a1c)
Δt=00:00:00.470

Keyword path:
- Tests
- section1
- L2 Power Headroom Reporting Onecell
- Power headroom reporting to PS with invalid order of the subframes, expected msg missing and buffering data
- user_setup.Create '65' users with cellid '${CELL_ID}'
- ${ueid} IN RANGE [1 | ${number of users}+1]

Figure 8. An overview of the results browser

The result browser was built with using HTML, SCSS (Sassy CSS), CSS (Cascading Style Sheets), javascript and jQuery. Sass (Syntastically Awesome StyleSheets) can be used to extend CSS with variables and imports among others. SCSS, the language of Sass, can be compiled to CSS. [61] jQuery is a JavaScript library which has been designed to make HTML object reading and writing, event handling, animations and AJAX (Asynchronous JavaScript And XML) operations simpler. Another advantage of jQuery is that it works on multiple browsers. [62]

The result browser was also developed using TDD. JavaScript library QUnit was selected as the testing framework because it provides reasonably good documentation and setting it up is quite easy. Tests can also be run quite fast and interactions can be tested automatically. [63] As an example of interaction testing, we have tests where clicking a certain object should set the class of another object correctly. Because the JavaScript for the results browser is written in object-oriented way, code can also be mocked quite easily by overriding methods to only save the parameters that they were called with.

In Figure 8, we can see that there are three main objects in the results browser. On the left there are a filtering tool and all the reported results that match the filters. By default, all repeats are visible. The filters include a maximum and minimum filters for length and repetition times (amount) of the repeats as well as msgId filter for removing repeats that contain certain msgId. There is also a reverse tool, search tool, in the filter tool for looking only repeats that contain certain msgIds. Filter tool is by default minimized and can be collapsed and minimized while retaining the data in it.

Repeats that pass filters are shown below the filtering tool in a table. The repeats can be sorted based on their length, repetition times (amount), average (AVG) duration and standard deviation (SD) of their durations. Each repeat is represented by the sequence of the msgIds in hexadecimal format in the message sequence.

Clicking on a repeat opens up "Specs view" object on the right where one can examine the repetition in more detail. In the Specs view, there are two objects: occurrences list and comparison tool. Each occurrence of the repeat is represented in the occurrence list. When first opening the Specs view, the comparison tool is hidden before two occurrences are selected from the occurrences list. In the comparison tool, both selected occurrences can be compared message by message. The same messages in each of the two occurrences are represented side by side.

All messages are minimized when the comparison tool is opened. Messages can be collapsed and minimized at will. A minimized message only shows the index of the message in the whole test run. Collapsed message shows the test case name, message name, msgId, keyword path and time delta (Δt) of the message. Time delta is the time taken since the first message in the occurrence. Therefore, for the first message in an occurrence, the following is always true: $\Delta t = 0$.

4.1.6. Analysis of repeats

As a basis of our case study analysis, we use a traced regression test run with 645 test cases where eight of them failed and rest of them passed. Four of the twelve sections contain failed tests cases and all tests passed in the other eight sections.

To get an overview of the results all supermaximal and pair-wise supermaximal repeats reported by the MRS Finder are copied to Microsoft Excel to make graphs about the amount of repeats and occurrences as well as average durations etc. We compare the results of supermaximal and pair-wise supermaximal repeats.

We also take a look at some specific pair-wise supermaximal repeats and analyze their occurrences with help from a case study software and SCT specialist. We start by sorting the results by decreasing average repeat duration in order to try to find the repeats that cost most to run. In this phase, we consider four filtering scenarios:

A. Minimum repeat amount 30
B. Minimum repeat amount 10 and minimum repeat length 30
C. Minimum repeat amount 10 and ignore some previously appeared msgIds
D. Ignore some previously appeared msgIds

For comparison, we continue the analysis by filtering out repeats that are only one message long and sorting the results by each field one at a time, thus considering top repeats in four sorting scenarios:

A. Sort by amount in decreasing order
B. Sort by length in decreasing order

C. Sort by average duration in decreasing order
D. Sort by duration standard deviation in decreasing order

For filtering scenarios, we pick repeats randomly from the top of the list for analysis, but for each sorting scenario, we pick the top five. We make notes about the repeats that are considered and present their characteristics in the results. The characteristics are tied into regression testing, test suite optimization, test case structure and mocking.

4.1.7. Implementation statistics measuring

Class and line coverage of Python unit tests among others can be measured using a tool called *nose* which can be installed using standard Python package installer, *pip* [64]. In this thesis, *nose* was used in a our continuous integration testing job to run the test, produce a coverage report and measure execution time of each unit test. Additionally, a Jenkins plugin called Cobertura Plugin was used to publish the results in readable format. From there, the results were copied to Microsoft Excel and formatted for more thesis suitable format. The continuous integration testing job was run on a machine with Intel Core 2 Duo and 4GB of RAM.

Suffix array creation is measured with 5730 messages long sequence. This sequence is assumed to be quite a bad case scenario with lot of repeats. We measure suffix array creation algorithm by measuring time it takes with the first 10, 100, 200, 300, 400, 500, 750, 1000, 1250, 1500, 1750, 2000, 2250, 2500, 2750, 3000, 3250, 3500, 3750, 4000, 4250, 4500, 4750, 5000, 5250, 5500 and 5738 messages from the sequence. We run this experiment on a laptop with Intel Core i5 and 4GB of RAM.

For the normal use case scenario, statistics are measured by running MRS Finder command line tool with 25, 30, 40, 50 and 70 threads with same data to find pair-wise supermaximal repeats as we use in the analysis. For comparison, we also run MRS Finder to find supermaximal repeats in one thread. These measures were run on the cloud computer with 30 cores and 2 TB of RAM.

4.2. Results

4.2.1. Implementation statistics

Figure 9 shows the line and class coverage of unit tests that were created during the development of the command line tool part of MRS Finder. It can be seen from the figure that the total class coverage is 100 percent while the total line coverage is over 90 percent.

The least covered module, msgid_plotter.py, is covered only 60 percent due to the fact that most functionality for the msgId plot is implemented in there. The remaining 40 percent of code include, for example, mocked parts of the UI that were tested manually.

The main module *find_same_message_sequences.py* has only 83 percent line coverage due to some parts being tested by launching the application in a subprocess. Code covered in a subprocess is not tracked in *nose*. This kind of testing is used mainly for command line option parsing and print outputs testing. This occurs also in *msg_parsing.py*.

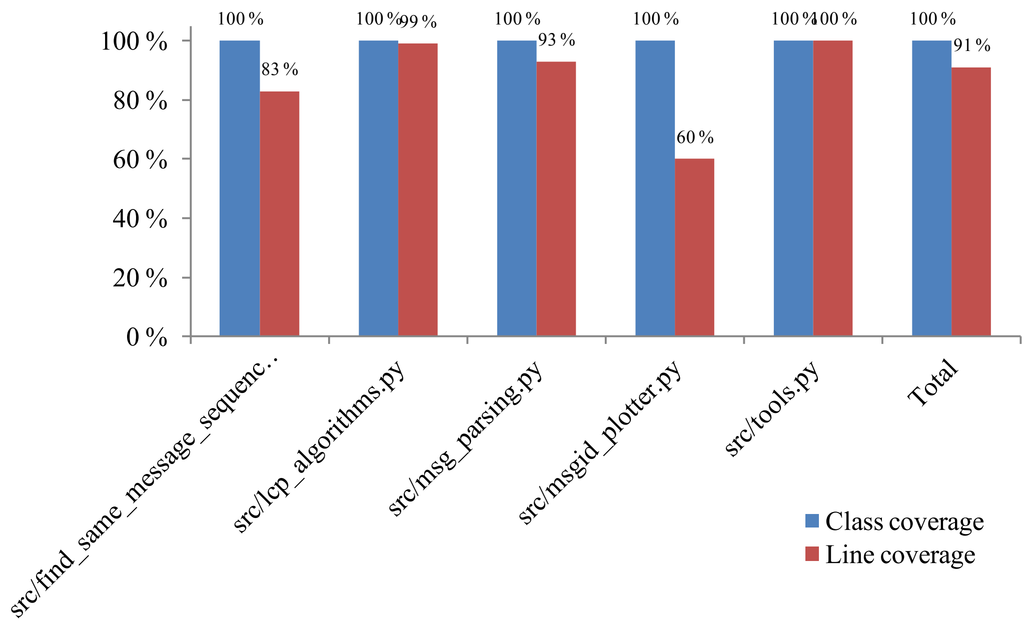


Figure 9. Line and class coverage of command line tool unit tests.

By considering all of the facts mentioned here, the total line coverage is actually even higher and probably quite close to 100%.

All 119 unit tests pass in the continuous integration job in 2 minutes and 9 seconds. This gives the average of one second per unit test. However, most of the time is spent on a one and half minute suffix array creation performance test which can be skipped during development by setting one boolean value to true if performance information is not required at the moment. Also it should be noted that 99 unit tests run under 10 milliseconds each.

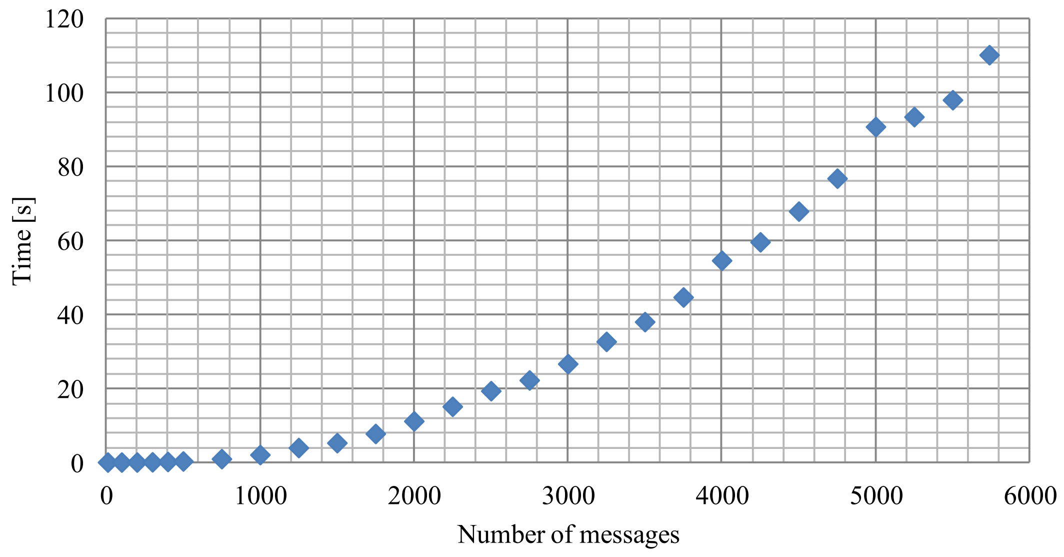
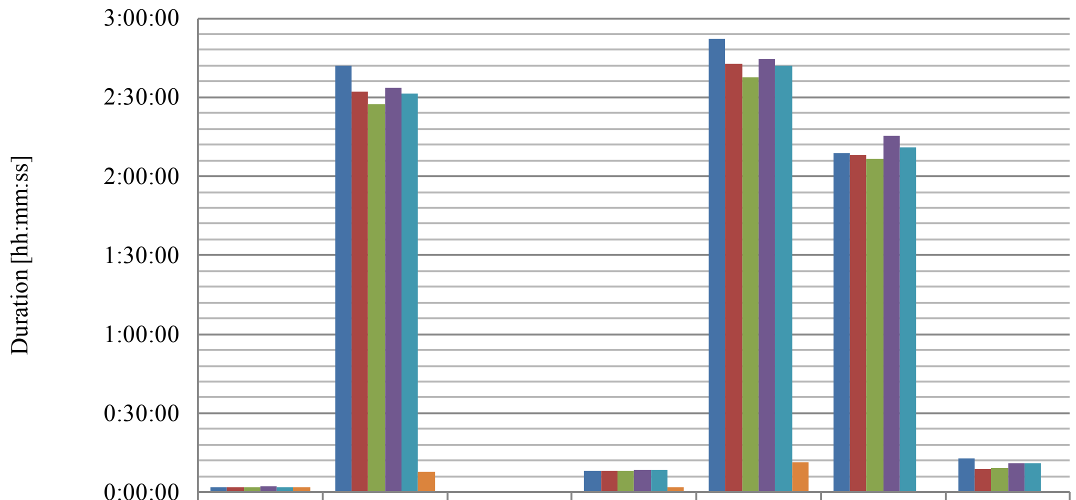


Figure 10. Duration measurements of suffix array creation algorithm with different amount of messages.

Suffix array creation measurements can be seen in Figure 10. The figure suggests that the implemented suffix array creation algorithm is not time linear but exponential. This is probably caused by the fact that suffixes are sorted with an algorithm where decision about which of two suffixes is larger is done by comparing messages one at a time in two suffixes and one character at a time in messages. Another factor is that a time linear suffix array creation algorithm was not used as a basis in the implementation.



	XML-parse	Threads/ Suffix arrays	Final duplicate removal	Finishing up	Total	Threads AVG	Threads SD
■ 25 threads	0:01:45	2:41:59	0:00:10	0:08:13	2:52:07	2:08:45	0:12:44
■ 30 threads	0:01:55	2:32:16	0:00:12	0:08:13	2:42:36	2:08:09	0:08:41
■ 40 threads	0:01:49	2:27:24	0:00:16	0:08:11	2:37:40	2:06:47	0:09:15
■ 50 threads	0:02:19	2:33:25	0:00:19	0:08:23	2:44:26	2:15:27	0:10:52
■ 70 threads	0:01:54	2:31:29	0:00:27	0:08:19	2:42:09	2:11:07	0:11:09
■ Supermaximals	0:01:49	0:07:35		0:01:56	0:11:20		

Figure 11. Duration measurements with different thread settings for pair-wise supermaximal repeats finding compared to duration measurements of supermaximal repeats finding. Note that XML-parse, duplicate removal and finishing up are actually not run in threads.

Figure 11 shows time measurements from the whole implementation. However, there are some notes that should be made before drawing any conclusions from it.

First, the XML-parse, final duplicate removal and finishing up phases are always done in one thread, the main thread. XML-parse is not affected by threads at all but the inputs of final duplicate removal and finishing up phases are. From the graph we can see that removing duplicates in the main thread gets slower as we add threads. This is probably because more duplicates are found between threads when there are more of them.

Finishing up phase should always get same input but it may be in different order with different amount threads because the workload is distributed pseudo-randomly. What finishing up phase does is that it formats the output in the correct format and writes the output files.

Second, supermaximal repeats are run only on the main thread due to technical reasons and cannot produce duplicates. Therefore, final duplicate removal phase is not applied to supemaximal repeats.

The first and most noticeable fact that can be seen in the figure is that the supermaximal repeats are tremendously cheaper to calculate than pair-wise supermaximals time-wise. It is faster to create one large suffix array of all messages than couple hundred thousand smaller suffix arrays with 30 of them in parallel at a given time. Another big factor is that the suffix arrays in thread take about two and a half hours even when run 30 in parallel.

Secondly, it can be seen that the time spent does not decrease much after 30 threads are reached because then the CPU will have to start scheduling the processes on its 30 cores. However, small improvements are possible with some extra threads because threads seem to end at different times. For example, when first thread of 30 threads ends then that core is not utilized any more. With 40 threads, 11 threads have to end before a core is not in full use. Therefore 40 seems to be the optimal amount of threads out of the measured thread amounts. The small increase in standard deviation of thread duration from 30 to 40 threads supports this.

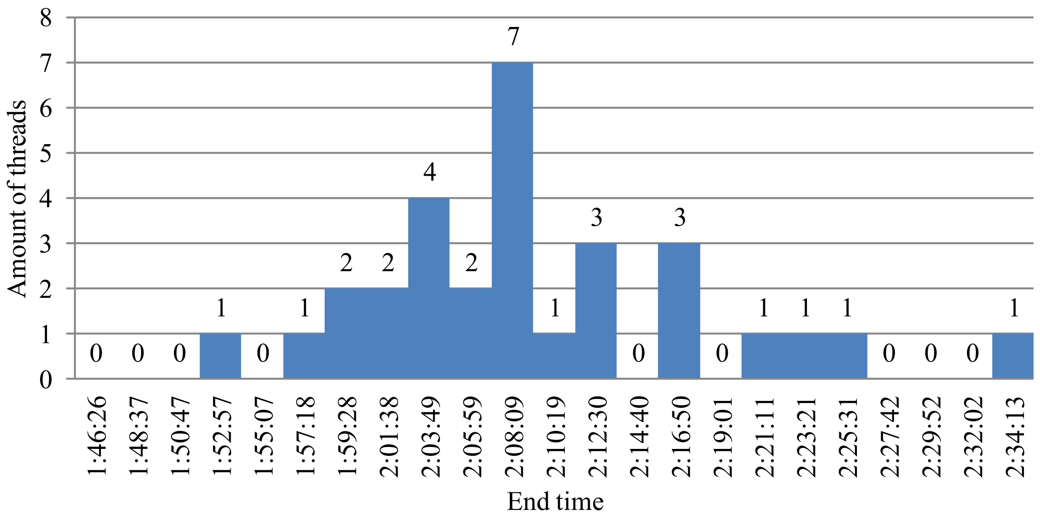


Figure 12. Distribution of thread durations with 50 threads on 30 cores. Average duration is 2:08:09 and standard deviation is 0:08:41.

The phenomenon of threads not ending at the same time is probably an effect of threads having different workloads because some suffix arrays are harder to create than others. The distribution of thread durations is visualized in Figure 12 for 30 threads. It seems that the last thread may end even almost an hour later than the first.

The total output file sizes of the command line part are 62.98 MB for supermaximal repeats and 63.56 MB for pair-wise supermaximal repeats. Most of the total file size consists of messages (62.56 MB) which is same for both.

4.2.2. Message ID plot

Message ID plot, which we refer to as msgId plot, shows the all traced messages as red balls in the order they appear in *output.xml*. The x-axis represents their ordinal number i.e. index and y-axis represents their msgIds in decimal format. A line from the bottom to top in the plot represents a beginning of a test case. Thus, a space between two lines represents one test case. The wider the space for the test case the more messages it has. The figure can be zoomed to any range. A full msgId plot is shown in Figure 13 and a couple of zoomed plots in Figure 14 and Figure 15.

The balls are interactive in the plot. Clicking on a ball shows the corresponding msgId and test case name on top of the plot. The click also produces a print in the console terminal where the MRS Finder was run from. In the print, there is additional information about the keyword path and the full message along with the information that appears also on top of the plot.

In Figure 13, the last ball is in x-axis at index 68010 which means that there are 68009 traced sent messages in this regression test because the first message has an index of zero. The msgIds range from 2139 to 60970.

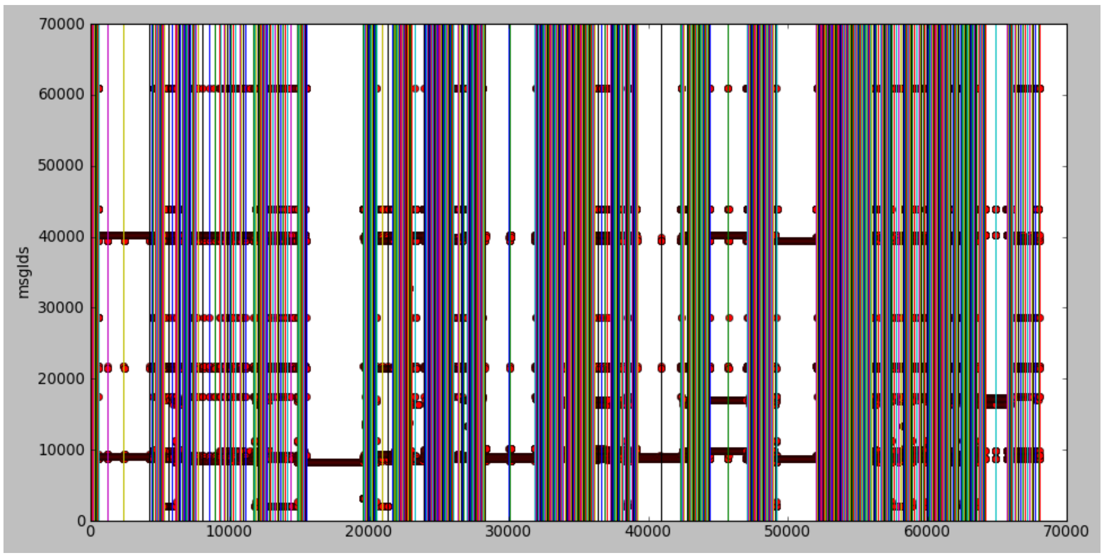


Figure 13. Full msgId plot of sent messages.

In Figure 14, the plot is zoomed so that the x-axis (indexes) now ranges from 58970 to 64461. In Figure 15, the range for x-axis is kept the same but the y-axis (msgIds) is also zoomed to range from 39403 to 40223. Thus, in the shown test cases, some of the messages that were visible in Figure 14 are not visible in Figure 15 because they are above or below the range.

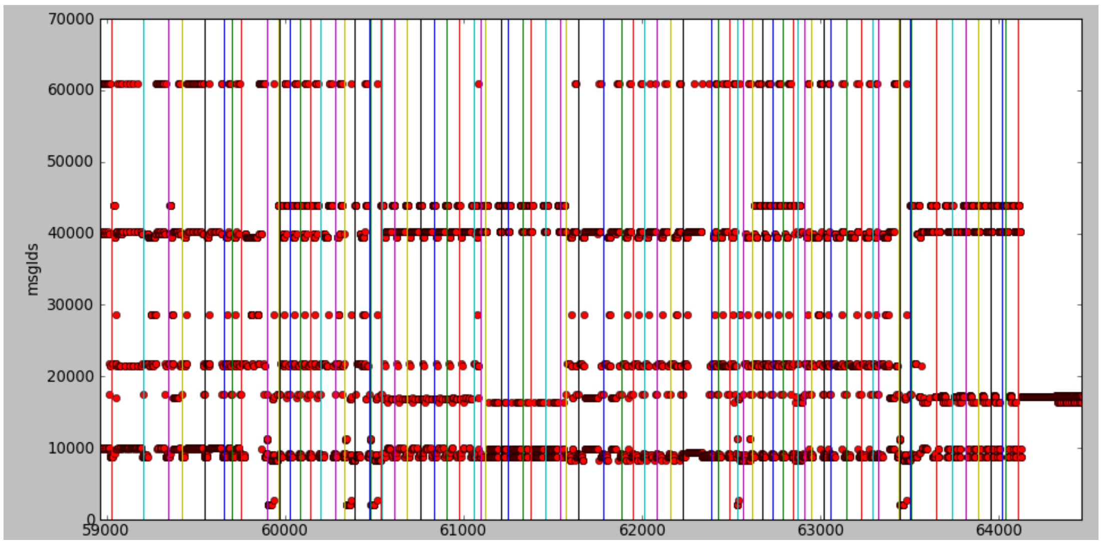


Figure 14. Zoomed msgId plot ranging from 58970 to 64461 in the x-axis.

Looking at the Figure 13 we see that some test cases have more traced messages than others and there are big ranges in y-axis where there are no messages. We can also see that there is couple of test cases that have more messages than others. Some small ranges in the y-axis seem to be used often, almost constantly, while other ranges are not used at all.

We see that there seems to be some repetition in the patterns of msgIds by looking at Figure 14. The same figure also indicates that one test case trace may contain messages with both very high and low value of msgId.

By looking at the Figure 15, we can see that msgIds, that appeared the same in Figure 14, may not be the same in fact. However, we can still see that there may be some repeated patterns.

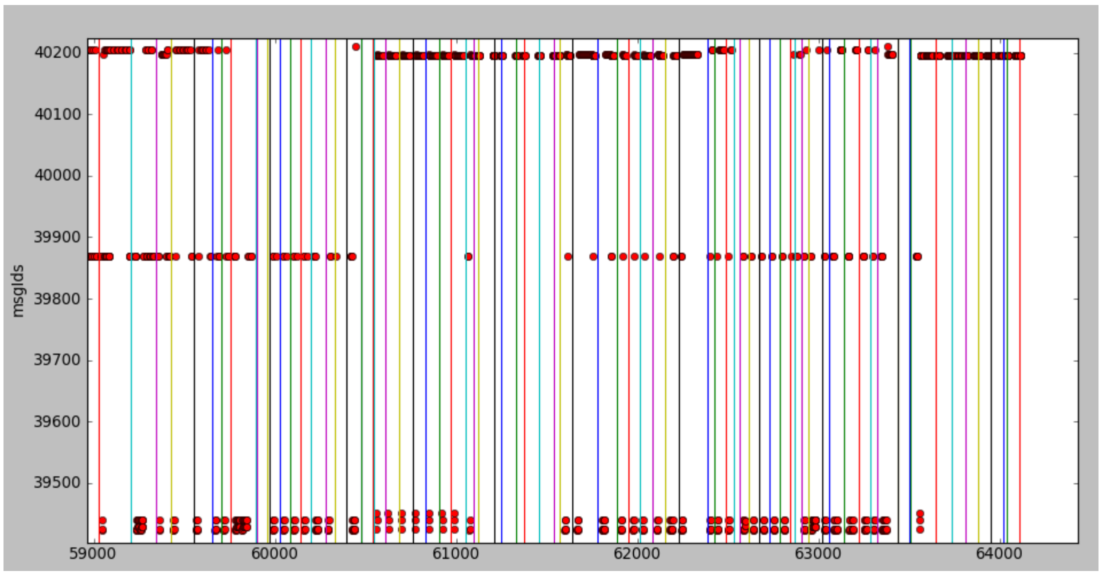


Figure 15. Zoomed msgId plot ranging from 58970 to 64461 in the x-axis and from 39403 to 40223 in the y-axis.

4.2.3. Overview of found repeats

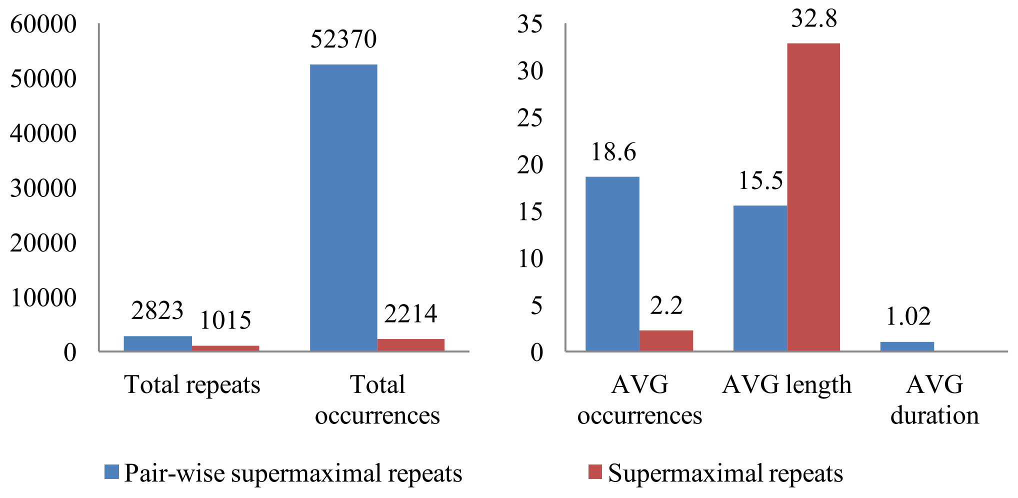


Figure 16. Comparison of pair-wise supermaximal repeats and supermaximal repeats

Figure 16 shows that overall that there are $\frac{2823}{1015} \approx 2.8$ times more pair-wise supermaximal repeats than supermaximal repeats in the test trace. By average, there are also $\frac{18.6}{2.2} \approx 8.5$ times more occurrences of a single repeat and, in total, $\frac{52370}{2214} \approx 23.7$ times more occurrences. Average repeat length is greater for supermaximal repeats by $\frac{32.8}{15.5} \approx 2.1$ times. Average duration for a repeat cannot be measured for supermaximal repeats reliably because the supermaximal repeats may occur between sections, but for pair-wise supermaximal repeats, it is about one second.

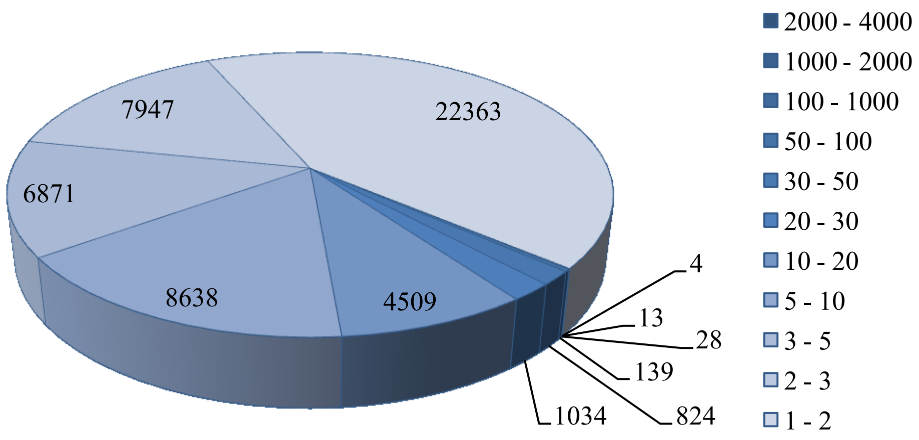


Figure 17. Amount of occurrences of pair-wise supermaximal repeats by their length. In total: 52370

In Figure 17, we can see the length distribution of occurrences for pair-wise supermaximal repeats. It seems that there are many occurrences that are quite short. Almost 75 percent of occurrences seem to have length of 4 or lower. However, there are thousands of occurrences that have length of over ten and hundreds of occurrences whose length is measured in tens, hundreds or even thousands.

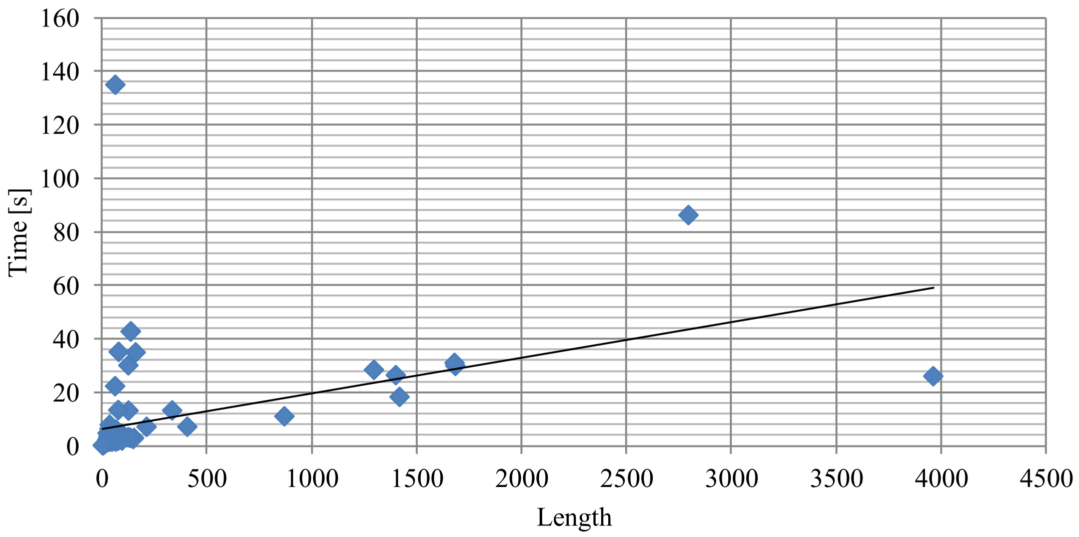


Figure 18. Average duration of pair-wise supermaximal repeats by their length

As the average length of the pair-wise supermaximal repeats is only 15.5, it can be quite hard to make linear plot of how a repeat length corresponds to duration. Either way, in Figure 18, we see that Microsoft Excel gives us a trend line that suggests that a longer repeat usually yields to a longer duration. However, we can see that there are many measurements that do not quite agree with the line. It should be noticed that the longest duration is not around 135 seconds because there are four repeats found for the length of 61 and the average of them is around 135 seconds. The longest average duration of a repeat is actually 531.310 seconds.

Figure 19 shows that there are less occurrences of supermaximal repeats than occurrences of pair-wise supermaximal repeats. It also shows that occurrences are distributed more on the longer end but there are not many very long occurrences.

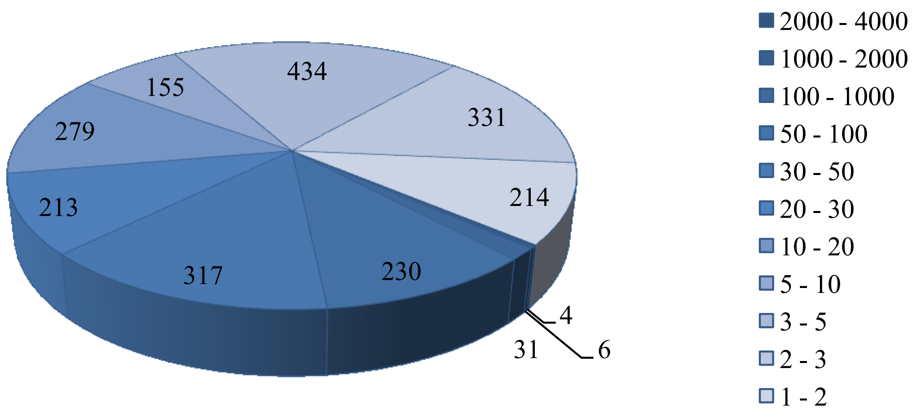


Figure 19. Amount of occurrences of supermaximal repeats by their length. In total: 2214

4.2.4. Analysis of specific repeats

In filtering scenario A, we find 261 pair-wise supermaximal repeats. Out of these repeats we find out that 28 of them contain one same msgId that is related to requesting a CPU load measurement from the test unit. Let's refer to this msgId later on as α. The repeat with the longest average duration (2.1 seconds) occurs in teardowns of 32 different test cases. For the case study SCT, this request is needed for checking some kernel errors in the test unit. This kind of measurement is usually also done in test cases that test capacity of the application. By looking at the other repeats that contain the same msgId it seems that there are over 300 test cases that do this check in their teardown.

Another sequence that is near the top of the list relates to checking status of resources and setting up test case environment in the test unit. This happens in hundreds of test cases. The repeat takes from half a second to three seconds every time to execute. In the case study SCT, it may be possible that these setups can be run only once at the start of a test suite instead of running it before each test case.

Following these two, there is a repeated sequence that relates to assertion phase of test cases and is about verifying correct data transmission. However the amount of data seems to be varying from test case to another, which may indicate that the message that is received could be varying too. There are also some near duplicates of this msgId sequence where the length is either longer or shorter.

Next up, there is a repeated sequence consisting of a test case setup which is used in multiple test cases that test same feature in different scenarios. This includes setting up mocks, measuring the CPU loads and setting some variables. This repeat occurs in 66 test cases and takes 0.773 seconds by average which means that the total time spent repeating it is about 51 seconds. This setup sequence is mandatory for some test cases but it should be investigated if it is used where it is not needed. Some optimizations for the repeat execution can also be possible. These settings may also be possible to set as default settings in initialization phase of a test suite rather than before each test case.

We also find a repeat which runs quite fast with duration of just over 0.1 seconds. The repeat is run on the action part of a test case. Any optimization benefits would be quite small with this repeat.

By looking quickly at the rest of the repeated sequences in scenario A, it is hard not to notice that most of the repeats are about setups or teardowns. There is also a lot of mock configuration which could probably be avoided by setting the mocks in the initialization phase with some default parameters of a test suite rather than setting them up every time in each test case setup, similarly to a repeat mentioned before.

Moving on to filtering scenario B, we only find 15 repeats and some of them seem to share same subsequences. First, we have a repeat that consists of regression test initialization phase in each section. Naturally, it occurs 12 times because of the 12 sections. By average it takes about a minute with a couple seconds of standard deviation. This part consists of more than sending messages and is often optimized in other ways in case study SCT.

Second, we have a repeat that contains previously mentioned msgId α again among others. The repeat occurs in a capacity test setup. The repeat takes 22.198 seconds in average and occurs in 12 test cases, which means that the total time spent in the repeat is well over four minutes. Some of the capacity test cases are requested by the customer of the developed product but a hypothesis is that some of the other capacity test cases may be redundant and can be removed. However, this information is not found in here. Either way, this should be looked into more closely.

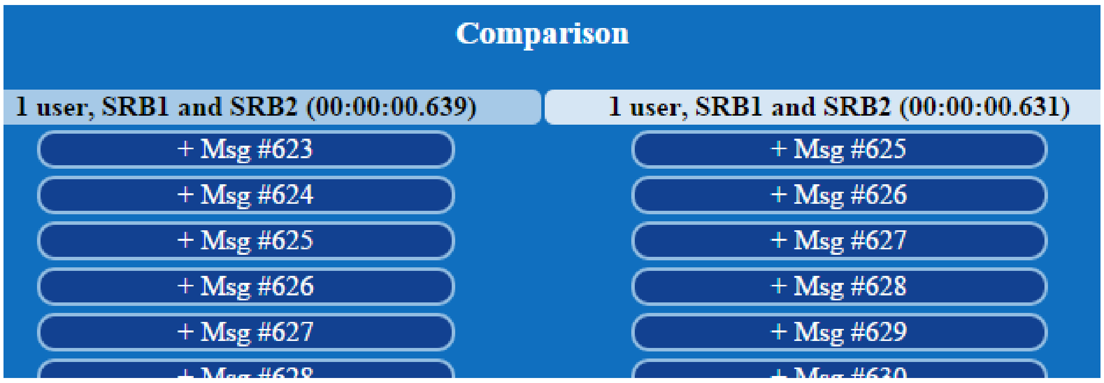
Third, we have yet another test setup related repeat. The repeat consists of smaller capacity test scenario setup and some CPU load measurement initialization. However, the occurrences of the repeat are not in the same test suite as the previous capacity tests. The repeat is 47 messages long, takes 3.357 seconds by average and occurs in 10 test cases yielding to 33.57 second total cost in the regression test.

Fourth, we notice a shorter version of the previous repeat which occurs in five more test cases and takes by average 2.577 seconds. That costs 38.655 seconds in total in the regression test. This repeat does not contain some of the CPU load measurement initialization and is only 32 messages long.

Rest of the repeats found in filtering scenario B are mostly about testing a certain feature in the case study software application. The feature is tested in multiple scenarios and the average duration of the repeat is about one second. Occurrences are within 10 to 18.

In filtering scenario C, we get quite short repeats in terms of average duration but we do get a lot of them, 371 to be more specific. The longest average duration is only 0.642 seconds and it is again about setting up the environment even though in the SCT Robot keyword it is explicitly said in the actual test part of each test case instead of their setup. Otherwise, the test cases seem different and should not be redundant with each other. Because the amount of occurrences is 11, the total time spent in the repeat is about 7 seconds.

Next, we have a repeat that is overlapping with itself in two test cases and therefore reported as 122 occurrences with a length of 42 messages. An example of two occurrences in comparison is presented in Figure 20. The test case in the figure is reported to have 120 occurrences where actually two different messages are repeated 262 times after each other. In the other test case the repeat is reported to occur two times meaning that the two messages are occur 22 times each.


Comparison

1 user, SRB1 and SRB2 (00:00:00.639)	1 user, SRB1 and SRB2 (00:00:00.631)
+ Msg #623	+ Msg #625
+ Msg #624	+ Msg #626
+ Msg #625	+ Msg #627
+ Msg #626	+ Msg #628
+ Msg #627	+ Msg #629

Figure 20. Overlapping occurrences of a repeat. Occurrence on the left contains messages #623 and #624 which are same as #625 and #626 as well as #627 and #628 respectively and so on.

After the overlapping repeat, we find four repeats more that are related to setting up test environment. Two of the repeats are almost the same with each other but differ by length of one message and the one with less messages appears in two test cases more. These four repeats occur 11 to 13 times and last less than half a second.

Next, we have a repeat of a teardown that occurs for all test cases in a certain suite but there are test cases in other suites where it occurs. The keyword name of the teardown changes between suites, which indicates that it has been implemented multiple times.

Then there is a repeat that occurs 27 times with length of 17 messages and average duration of 0.404 seconds, which adds to 10.908 seconds in total. The test cases where the repeat occurs are created with a test case template. The repeat occurs at setup phase of the test cases.

For the rest of the repeats in filtering scenario C, there are various repeats that seem familiar to the already inspected but are bit shorter. Most of them also seem to relate to test case setup at first glance.

On the final filtering scenario D, we get even more repeats (1707) by removing the minimum amount filter. First up, there is an overlapping repeat where a two message long repeat occurs only in one test case. The repeat happens in the action part of the test case. The test case tests data transferring and the same data being sent over and over is visible here.

Next from the top, we have a repeat that occurs three times and is 1678 messages long with an average duration of 30.846 seconds. This means that more than a minute and a half is consumed in total. The repeat consists of deleting users from the application in teardown phase. This is done with a certain Robot Framework keyword in a loop but in the case study software application there is a faster way for deleting users. Thus this repeat can be optimized. Two repeats after it are about the same repeat but in different versions appearing in longer and shorter versions where they have one less and one more occurrences respectively. The one with four occurrences takes one minute and 45 seconds in total.

Moving forward, we have an overlapping repeat in a test case where 20 messages are repeated over and over. It is a logging level test. In case study CI, a lot of tests use logging information and therefore they should fail if logging breaks. The same msgIds as in the repeat can also be found in numerous other repeats by using the search tool. The bad thing is that the logging test takes about 25 seconds to execute and in the actual Robot test case it literally reads: repeat a keyword 100 times. From the test result database, we can see that the test has not been broken in a while.

Next up, there is another overlapping repeat that occurs in one test case. This time though, 353 messages are repeated over and over. The repeat resides in the action part of the test case where data is transmitted between the couple of mocks and the test unit. The repeat is reported to have duration of about 18 seconds.

The sorting scenario A is very hard to analyze because the amount of occurrences is quite high and it would take very long to go through each occurrence. However, the common factor for the top five repeats seems to be that they are quite short and occur in many different teardowns and setups. The top five is shown in Figure 21.

	Amount	**Length**	**Duration AVG**	**Duration SD**
1st	739	6	0.033	0.004
2nd	688	6	0.032	0.005
3rd	613	2	0.158	0.072
4th	609	2	0.014	0.003
5th	414	2	0.117	0.060

Figure 21. Top five repeats with most occurrences (amount) that are longer than one (sorting scenario A)

Now in the sorting scenario B, we see that there are a few quite long repeats that consist of thousands of messages. However the first, second and fifth are overlapping repeats that only occur in one test case. The first repeat occurs in the logging test an the repeat was already mentioned previously. The second one occurs in a stability test that does the same thing 800 times. The third, fourth and fifth have already been analyzed previously. The top five is presented in Figure 22.

	Amount	**Length**	**Duration AVG**	**Duration SD**
1st	2	3960	25.861	0.021
2nd	2	2793	86.068	0.005
3rd	2	1682	29.703	0.180
4th	3	1678	30.846	2.108
5th	2	1416	18.127	0.057

Figure 22. Top five longest repeats by length (sorting scenario B)

Figure 23 shows the top five test cases in sorting scenario C. We see that the longest maximal repeat by average duration is over 8 minutest long. However this repeat is a section suite initialization and section suite teardown combined. The last message in the repeat is the last message in the section which appears in the Robot Framework output.xml before the actual tests appear. For this reason, an occurrence of the repeat has the duration of a whole section but does not include any of the messages in the test cases of the section. This repeat appears only eight times because different section teardown is run if all tests pass in a section. We can see that this repeat occurs in sections where all tests passed.

The second on the list is the same repeat as in the sorting scenario B. Third and fourth occur in all section initializations but the fourth also appears in a test case where a crash of the case study software application is made to happen and the test unit is then retrieved back up with section initialization. The retrieving part of the test case matches a part of the section initialization.

The fifth repeat occurs in two capacity test cases. The repeat starts almost at the beginning keywords of the test cases and ends in teardown. This is the first analyzed repeat that seems to cover almost in full two test cases from start to beginning. However the repeat does not cover test case setup phases of the test cases.

	Amount	**Length**	**Duration AVG**	**Duration SD**
1st	8	61	531.310	115.270
2nd	2	2793	86.068	0.005
3rd	12	60	60.691	2.019
4th	13	24	58.512	1.807
5th	2	135	42.683	0.037

Figure 23. Top five longest repeats by average duration with length more than one (sorting scenario C)

The top five repeats in sorting scenario D are shown in Figure 24. In the very top of the table, we have the same repeat as in the sorting scenario C. Second, third and fourth repeat occur in capacity cases where the CPU loads are measured. Such repeats have been analyzed before in the filtering scenarios. The fifth repeat occurs in test cases using a same data transfer template. However, the data has been sent before the repeat starts. Based on this evidence, the test cases do not seem to send same data and are not redundant in this matter.

	Amount	**Length**	**Duration AVG**	**Duration SD**
1st	8	61	531.310	115.270
2nd	13	32	22.426	5.791
3rd	12	33	25.198	4.455
4th	9	33	25.555	4.331
5th	3	7	7.375	4.113

Figure 24. Top five repeats by duration standard deviation with length more than one (sorting scenario D)

5. DISCUSSION

5.1. Challenges of MRS Finder

During the analysis, it was found that msgId plot is quite hard to use. First of all, before zooming in, it shows a plot with a lot of data in it. UI-wise, it seems just too much at first glance. Secondly, when viewing the full plot, it seems that there are a lot of traced messages in a row which have the same msgId. However, when we zoom in, it can be seen that this is not the case. This is due to the fact that msgIds seem to cover only small areas of the available msgId space. They are concentrated on some smaller ranges. Thirdly, even if two msgIds were the same, it does not necessarily mean that the two corresponding messages would be the same as stated in chapter 4.1.3. All in all, the main benefit of msgId plot seems to be that it gives hope that there may be redundancy in the traces.

The command line tool of MRS Finder has some performance issues and requires a cloud computer with many cores and a lot of RAM when searching for pair-wise supermaximal repeats. In chapter 4.2.1, it was presented that the algorithm does not have linear time complexity. However, in the case study CI, this is acceptable because MRS Finder can be run during the night. However, it would be more ideal if it could be run after each regression test so that every regression test could be analyzed. One of the bright things about the application is the execution time of supermaximal repeats search. However, the main point was to look for pair-wise supermaximal repeats which was now only done for sent messages mainly because of the performance issues.

The results browser part works fluently performance-wise when the results have been loaded into browsers memory which is not a problem in the internal network of the case study company. However, UI-wise there is room for improvement. During the analysis of the results, we had to go look at the Robot keyword code to see at which point in the test case the repeat occurs. Straight from the result browser, it was not immediately clear in some cases. It was also not clear if the repeat covered the whole test case or not. The question was that did the test case contain some messages before the first message in the repeat or after the last message. Another problem was that the already analyzed repeats kept polluting the results by appearing again in different filter and sorting settings.

5.2. Discovered repeats

It was discovered that there are quite a low amount of occurrences of each supermaximal repeat. It is also evident that supermaximal repeats are long and may overlap each other, which makes the analysis harder. In the results it was also shown that we get much more repeats and occurrences of pair-wise supermaximal repeats than supermaximal repeats. Some of them are also quite long but there are many more occurrences of repeats with medium and long length. We had some good finds which probably would not have found with supermaximal repeats because supermaximal repeats do not, first of all, occur so many times.

One problem with pair-wise supermaximal repeats was that we also found many repeats that were near duplicates of each other having mutations of changed, removed or added messages. These repeats polluted the results and made finding other repeats harder. On the other hand, it was also a good feature because we could see that the same repeat appeared in multiple test cases and varied just a little in them, pointing out the difference.

Another problem was that we were not able to find a repeat that would cover a full test case. At least we were not able to prove it. The repeats also tended to occur mostly in setup and teardown phases of the test cases, which actually makes sense because test suites are a logical collection of tests and therefore often share same test case setup and teardown.

Overlapping was also another problem as with supermaximal repeats. We may have still not found all repeats that we would have liked because message sequence appearing over and over is reported as only two occurrences. When comparing this kind of repeat with smaller repeat of the same sequence, only the larger and overlapping is reported.

Overall, the found repeats provide good overview on where there is repetition in the test trace. The repeats also revealed some badly coded and slow tests which can now be refactored. The refactoring process can also be prioritized by the average duration of the found repeats.

5.3. Trace repeat detection compatibility with test suite optimization

In my opinion, the results of the command line part of MRS Finder enable test suite optimization in theory because we can now link test cases by their similarities. However in practice, I do believe that no matter which method was used, the results of test suite optimization would not be good for numerous reasons. We have mostly repeats that only occur in setups and teardowns. Also there are many repeats that are mostly the same and link to the same requirement. Because of this some kind filtering system would have to be implemented to remove some of the repeats. They cannot be filtered manually because there is so much repetition.

Actually, MRS Finder seems to be better tool for another kind of optimization with its results browser. It is now possible to see what sequences are repeated a lot and which repeats take time. It is possible to optimize these repeats to take less time manually and move some test setups to higher level to run only once if they take so much time. Overall, the MRS Finder makes some bad code more visible. Nevertheless, the result of this study is negative in the matter of trace repeat detection compatibility with test suite optimization

In the CI process, the tool suites well to a situation where the regression test suite should be optimized but no-one knows the tested functionality of each regression test case. The tool can then be run to inspect what functionality is repeated often and which of the repeats take the most time to execute. The tool should be used manually by developers who are familiar with the functionality of the tested application. On the other hand, execution of MRS Finder should be automatic. The tool should be run as often as possible after new regression test results are available. This would keep the results of MRS Finder up-to-date.

5.4. Future work

MRS Finder needs performance improvements in future because the amount of test cases grows with the time and so does the amount of traced messages. The performance can possibly be increased by improving suffix array creation algorithm. The literature shows that the implemented sorting method is not the fastest and it can possibly be done in linear time. It also may not be possible because in the case study, we have complete strings instead of single characters. It can be looked into, however. Another performance increase would be creating a PLCP table in order to compute the LCP table faster. Other improvement methods should be investigated as well like writing the algorithm in C.

More analysis should be also done from the results of MRS Finder. We only looked at very small percentage of the reported repeats and there may still be more good and interesting finds. We also run MRS Finder against a trace of a single regression test run and the results can be quite different with other regression tests.

Improvement possibilities to the results browser UI include adding more filters, creating better links to Robot Framework keyword code and showing messages in teardown and setups different color, for example, in order to differentiate them from the repeat. Filters could be created for already analyzed repeats, minimum and maximum duration as well as for resource files. With a resource file filter, a developer optimizing a Robot keyword resource file could see what part of it is used heavily and see if the repeats are improved over time.

New aspects that could be investigated and would interest me include showing the messages that differ in the results browser, a sequence alignment of test cases and categorizing repeats. Repeats could be categorized if they share some threshold volume of same messages. This could either be shown in the results browser or used in possible future test suite optimization. Different type of repeats could be also tried.

Finally, the method of removing test cases with their complete trace occurring in a prefix of another test case could be tried. This method was presented in chapter 2.2.3. However, based on our results, it seems that no such test case would be found where this would happen.

6. CONCLUSION

Large scale continuous integration and regression testing can be rather problematic because the testing tends to take time and resources which are often limited. Complete re-test all tests method becomes impossible because the feedback time from the commits increases.

This thesis studied many test suite optimization methods that have been used to reduce the test suite by removing tests and automatically creating new tests. All of them need the knowledge of which test case overlaps which one. This information is usually either code coverage or requirement coverage of a single test case. It was suggested that test cases that share same coverage as other test cases do not have to be run. And the test cases that have maximum coverage with smallest amount of cost should be run first.

This thesis presented something new. We looked into several methods of finding repeats on strings. It turned out that the field of finding repeats and comparing strings has many problems and solutions. It was not found that anyone had tried to find supermaximal repeats their test trace before. The concept of pair-wise supermaximal repeat was also introduced. We considered a new suffix array method where a suffix was considered as a list of strings instead of just a string.

In our case study, we had traced messages in test cases which often repeated themselves. A custom enhanced suffix array algorithm was implemented to search for pair-wise supermaximal repeats, supermaximal repeats that occur in a concatenation of two test case traces. A new application was developed: MRS Finder. MRS Finder can use multiple cores at the same time to compare to create suffix arrays and find pair-wise supermaximal repeats in them. However, MRS Finder struggles with suffix array creation with long message sequences and leaves room for optimization in the application.

To analyze the findings of MRS Finder, the pair-wise supermaximal repeats that were found from a regression test were compared to supermaximal repeats which can also be found with the same implementation. Some interesting individual repeats were also picked and analyzed further.

It turned out that most pair-wise supermaximal repeats occur in test setup or teardown phases of the test cases. No test case was proven to be completely redundant to another. However, a huge amount of repetition was found on the test trace and MRS Finder turned out to be a good test optimization tool for searching parts in test code that could be optimized while maybe not being a good tool for test suite optimization. For the case study CI process, this means that developers now have a new tool for refactoring their tests to make them run faster in CI. The tool also makes bad code more visible and shows that there are plenty of ways to improve test code.

There is also a lot of room for more work in this field. Sequence alignment of test case traces, categorizing repeats, different type of repeats and filters should be tried.

7. REFERENCES

[1] Graham D., van Veenendaal E., Evans I. & Black R. (2008) Foundations of Software Testing: ISTQB Certification. Cengage Learning EMEA, London, UK.

[2] Rogers, R.O. (-p) Scaling Continuous Integration. None: 68-76. DOI:http://dx.doi.org/10.1007/978-3-540-24853-8_8.

[3] Prasad, S., Jain, M., Singh, S. & Patvardhan, C. (2012) Regression Optimizer A Multi Coverage Criteria Test Suite Minimization Technique. International Journal of Applied Information Systems 1(8): 5-11. DOI:http://dx.doi.org/10.5120/ijais12-450215.

[4] Malhotra, R., Kaur, A. & Singh, Y. (2010) A Regression Test Selection and Prioritization Technique. Journal of Information Processing Systems 6(2): 235-252. DOI:http://dx.doi.org/10.3745/JIPS.2010.6.2.235.

[5] Kim, J-M. & Porter, A. (2002) A history-based test prioritization technique for regression testing in resource constrained environments. Proceedings of the 24th International Conference on Software Engineering: 119-129. DOI:http://dx.doi.org/10.1145/581339.581357.

[6] Marijan, D., Gotlieb, A. & Sen, S. (2013) Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. 29th IEEE International Conference on Software Maintenance: 540-543. DOI:http://dx.doi.org/10.1109/ICSM.2013.91.

[7] Response to Stop 29119 Petition (2014) URL: http://www.softwaretestingstandard.org/29119petitionresponse.php. Accessed: 10.4.2015.

[8] Stop 29119 (2014) URL: http://commonsensetesting.org/stop29119/. Accessed: 10.4.2015.

[9] Robot Framework User Guide 2.8.7 (2015) URL:http://robotframework.org/robotframework/2.8.7/RobotFrameworkUserGuide.html. Accessed 8.4.2015.

[10] Amiar, A., Delahaye, M., Falcone, Y. & du Bousquet, L. (2013) Fault localization in embedded software based on a single cyclic trace. 24th International Symposium on Software Reliability Engineering: 148-157. DOI:http://dx.doi.org/10.1109/ISSRE.2013.6698914.

[11] Gang, Y., Xianjun, L. & Zhongwen, L. (2011) Fault localization with intersection of control-flow based execution traces. 3rd International Conference on Computer Research and Development: 430-434. DOI:http://dx.doi.org/10.1109/ICCRD.2011.5764051.

[12] Cormen, T.H., Leiserson, C.E., Rivest, R.L. & Stein, C.(2009) Introduction to algorithms, third edition. Cambridge, MA.

[13] Russenschuck, S. (1994) Techniques and applications of mathematical optimization and synthesis. IEE Colloquium on Optimisation and Synthesis of Electromagnetic Fields: 1-3

[14] Leandro, S.P., Saurabh, S. & Alessandro, O. (2012) Understanding myths and realities of test-suite evolution. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering: 33:1-33:11. DOI:http://dx.doi.org/10.1145/2393596.2393634.

[15] Harrold, M.J., Gupta, R. & Soffa, M.L. (1990) A methodology for controlling the size of a test suite. Proceedings of Conference on Software Maintenance: 302-310. DOI:http://dx.doi.org/10.1109/ICSM.1990.131378.

[16] Xu, S., Miao, H. & Gao, H. (2012) Test Suite Reduction Using Weighted Set Covering Techniques. 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing : 307-312. DOI:http://dx.doi.org/10.1109/SNPD.2012.87.

[17] Zhang, Y., Liu, J., Cui, Y. & Hei, X. (2011) An improved quantum genetic algorithm for test suite reduction. IEEE International Conference on Computer Science and Automation Engineering 2: 149-153. DOI:http://dx.doi.org/10.1109/CSAE.2011.5952443.

[18] Anwar, Z. & Ahsan, A. (2014) Exploration and analysis of regression test suite optimization. ACM SIGSOFT Software Engineering Notes 39(1): 1-5

[19] Garey, M.R. & Johnson, D.S. (1976) Some simplified NP-complete graph problems. Theoretical Computer Science 1(3): 237–267. DOI:http://dx.doi.org/10.1016/0304-3975(76)90059-1 .

[20] Jeffrey, D. & Gupta, R. (2005) Test suite reduction with selective redundancy. Proceedings of the 21st IEEE International Conference on Software Maintenance: 549-558. DOI:http://dx.doi.org/10.1109/ICSM.2005.88.

[21] Wong, W.E., Horgan, J.R., London, S. & Agrawal, H. (1997) A study of effective regression testing in practice. Proceedings of the 8th IEEE International Symposium on Software Reliability Engineering: 264-274. DOI:http://dx.doi.org/10.1109/ISSRE.1997.630875.

[22] Mansour, N. & Statieh, W. (2009) Regression test selection for c# programs. Advances in Software Engineering: 1:1-1:16. DOI:http://dx.doi.org/10.1155/2009/535708.

[23] Inozemtseva, L. & Holmes, R. (2014) Coverage is not strongly correlated with test suite effectiveness. Proceedings of the 36th International

Conference on Software Engineering: 435-445. DOI:http://dx.doi.org/10.1145/2568225.2568271.

[24] Tallam, S. & Gupta, N. (2005) A concept analysis inspired greedy algorithm for test suite minimization. Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering: 35-42. DOI:http://dx.doi.org/10.1145/1108792.1108802.

[25] Jeffrey, D. & Gupta, R. (2007) Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. IEEE Transactions on Software Engineering 33(2): 108-123. DOI:http://dx.doi.org/10.1109/TSE.2007.18.

[26] Nanda, A., Mani, S., Sinha, S., Harrold, M.J. & Orso, A. (2011) Regression testing in the presence of non-code changes. IEEE Fourth International Conference on Software Testing, Verification and Validation: 21-30. DOI:http://dx.doi.org/10.1109/ICST.2011.60.

[27] Sun, C. (2011) A Constraint-based Test Suite Reduction Method for Conservative Regression Testing. Journal of Software 6(2): 314-321. DOI:http://dx.doi.org/10.4304/jsw.6.2.314-321.

[28] Benala, T.R., Dehuri, S. & Mall, R. (2012) Computational intelligence in software cost estimation: an emerging paradigm. ACM SIGSOFT Software Engineering Notes 37(3): 1-7. DOI:http://dx.doi.org/10.1145/180921.2180932.

[29] Zheng Li., Harman, M. & Hierons, R.M. (2007) Search Algorithms for Regression Test Case Prioritization. IEEE Transactions on Software Engineering: 225-237. DOI:http://dx.doi.org/10.1109/TSE.2007.38.

[30] Qian, Z. (2010) Test Case Generation and Optimization for User Session-based Web Application Testing. Journal of Computers 5(11): 1655-1662. DOI:http://dx.doi.org/10.4304/jcp.5.11.1655-1662.

[31] Hamer, M. & Ortega-Sanchez, C. (2010) Simulation platform for the evaluation of robotic swarm algorithms. IEEE Region 10 Conference: 1583-1588. DOI:http://dx.doi.org/10.1109/TENCON.2010.5686044.

[32] De Souza, L.S., de Miranda, P.B.C., Prudencio, R.B.C. & de Barros, F.A. (2011) A Multi-objective Particle Swarm Optimization for Test Case Selection Based on Functional Requirements Coverage and Execution Effort. 23rd IEEE International Conference on Tools with Artificial Intelligence: 245-252. DOI:http://dx.doi.org/10.1109/ICTAI.2011.45.

[33] Kaur, A. & Bhatt, D. (2011) Particle Swarm Optimization with Cross-Over Operator for Prioritization in Regression Testing. 27(10): 27-34. DOI:http://dx.doi.org/10.5120/3336-4589.

[34] Zadeh, L.A. . (2008) Is there a need for fuzzy logic?. Annual Meeting of the North American Fuzzy Information Processing Society: 1-3. DOI:http://dx.doi.org/10.1109/NAFIPS.2008.4531354.

[35] Ramot, D., Friedman, M., Langholz, G. & Kandel, A. (2003) Complex fuzzy logic. IEEE Transactions on Fuzzy Systems 11(4): 450-461. DOI:http://dx.doi.org/10.1109/TFUZZ.2003.814832.

[36] Mendel, J.M. (1995) Fuzzy logic systems for engineering: a tutorial. Proceedings of the IEEE : 345-377. DOI:http://dx.doi.org/10.1109/5.364485.

[37] Haider, A.A., Rafiq, S. & Nadeem, A. (2012) Test suite optimization using fuzzy logic. International Conference on Emerging Technologies: 1-6. DOI:http://dx.doi.org/10.1109/ICET.2012.6375440.

[38] Gusfield, D. (1997) Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press. New York, NY.

[39] Babenko, M.A. & Starikovskaya, T.A. (2008) Computing Longest Common Substrings Via Suffix Arrays. Computer Science – Theory and Applications: 64-75. DOI:http://dx.doi.org/10.1007/978-3-540-79709-8_10.

[40] Rubi, R.D. & Arockiam, L. (2012) Positional_LCS: A position based algorithm to find Longest Common Subsequence (LCS) in Sequence Database (SDB). IEEE International Conference on Computational Intelligence & Computing Research: 1-4. DOI:http://dx.doi.org/10.1109/ICCIC.2012.6510271.

[41] Huang, X. & Chao, K.M. (2003) A generalized global alignment algorithm. Bioinformatics 19(2): 228-233. DOI:http://dx.doi.org/10.1093/bioinformatics/19.2.228.

[42] Davidson, A. (2001) A fast pruning algorithm for optimal sequence alignment. Proceedings of the IEEE 2nd International Symposium on Bioinformatics and Bioengineering Conference: 49-56. DOI:http://dx.doi.org/10.1109/BIBE.2001.974411.

[43] Elhadi, M. & Al-Tobi, A. (2009) Webpage Duplicate Detection Using Combined POS and Sequence Alignment Algorithm. WRI World Congress on Computer Science and Information Engineering: 630-634. DOI:http://dx.doi.org/10.1109/CSIE.2009.771.

[44] Rashid, N.A.A., Abdullah, R., Talib, A.Z.H. & Ali, Z. (-p) Fast Dynamic Programming Based Sequence Alignment Algorithm. None: 1-7. DOI:http://dx.doi.org/10.1109/DFMA.2006.296909.

[45] Daugelaite, J., O' Driscoll, A. & Sleator, R.D. (2013) An Overview of Multiple Sequence Alignments and Cloud Computing in Bioinformatics.

International Scholarly Research Notices Biomathematics 2013: 615630:1-615630:14. DOI:http://dx.doi.org/10.1155/2013/615630.

[46] Abouelhodaa, M.I., Kurtzb, S. & Ohlebuscha, E. (2004) Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2(1): 53–86. DOI:http://dx.doi.org/10.1016/S1570-8667(03)00065-0.

[47] Applications of Suffix Trees (2004) URL: http://www.cs.ucf.edu/courses/cap5937/fall2004/Applications%20of%20suffix%20tree2.ppt. Accessed: 10.4.2015.

[48] Manber, U. & Myers, G. (1990) Suffix arrays: a new method for on-line string searches. Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms: 319-327

[49] Abouelhoda, M.I. & Dawood, A. (2008) Fine Tuning the Enhanced Suffix Array. Cairo International Biomedical Engineering Conference: 1-4. DOI:http://dx.doi.org/10.1109/CIBEC.2008.4786047.

[50] Shun, J. (2014) Fast Parallel Computation of Longest Common Prefixes. International Conference for High Performance Computing, Networking, Storage and Analysis: 387-398. DOI:http://dx.doi.org/10.1109/SC.2014.37.

[51] Kärkkäinen, J., Manzini, G. & Puglisi S.J. (2009) Permuted Longest-Common-Prefix Array. Combinatorial Pattern Matching, Lecture Notes in Computer Science 5577: 181-192. DOI:http://dx.doi.org/10.1007/978-3-642-02441-2_17.

[52] Gog, S. & Ohlebusch, E. (2011) Fast and Lightweight LCP-Array Construction Algorithms. Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments: 25-34. DOI:http://dx.doi.org/10.1137/1.9781611972917.3.

[53] Beck, K. (2003) Test-driven Development: by Example. Addison-Wesley Longman.

[54] Kim, T., Park, C. & Wu, C. (-p) Mock Object Models for Test Driven Development. None: 221-228. DOI:http://dx.doi.org/10.1109/SERA.2006.49.

[55] Jenkins Wiki (2014) URL: https://wiki.jenkins-ci.org/. Accessed: 27.3.2015.

[56] Robot Framework (2015) URL: http://robotframework.org/. Accessed: 10.4.2015.

[57] Unittest - Python 2.7.10rc0 documentation(2015) URL: https://docs.python.org/2/library/unittest.html. Accessed: 10.4.2015.

[58] Python 2.7.10rc0 documentation (2015) URL: https://docs.python.org/2/. Accessed: 10.4.2015.

[59] Matplotlib 1.4.3 documentation (2015) URL: http://matplotlib.org/. Accessed: 10.4.2015.

[60] Multiprocessing - Python 2.7.10rc0 documentation (2015) URL: https://docs.python.org/2/library/multiprocessing.html. Accessed: 10.4.2015.

[61] Sass Documentation (2015) URL: http://www.sass-lang.com/documentation/file.SASS_REFERENCE.html. Accessed: 10.4.2015.

[62] jQuery API Documentation (2015) URL: http://api.jquery.com/. Accessed: 10.4.2015.

[63] Introduction to Unit Testing - QUnit (2012) URL: http://qunitjs.com/intro/. Accessed: 10.4.2015.

[64] Nose 1.3.6 documentation (2015) URL: https://nose.readthedocs.org/en/latest/. Accessed: 10.4.2015.