# Performance of near-duplicate detection algorithms for Crawljax

Erwin van Eyk Wilco van Leeuwen

Delft University of Technology
{E.D.C.vanEyk, W.J.vanLeeuwen}@student.tudelft.nl

Coach: Arie van Deursen
Client: Alex Nederlof

Bachelor Project Coordinator:
Martha A. Larson and Felienne Hermans

May 2014

# 1 Preface/Foreword

This report contains the documentation of the Bachelor of Science final project of Erwin van Eyk and Wilco van Leeuwen. This project is the final stage of the Bachelor of Science from the study Computer Science at the Delft University of Technology (TU Delft). The project is concerned with Crawljax, an open source project maintained by TU Delft and the University of British Colombia. Crawljax is a JavaScript-enabled web crawler that can automatically discover user interface states of a web page. The goal of this project was to detect near-duplicate states automatically during a crawl. In this document we will explain the different approaches we used to tackle this problem, the way we tested the algorithms and the decisions we made during the project.

# Contents

# 2 Summary

Crawljax is a crawler, which not only finds states via regular links, but also states that are hidden by JavaScript actions. However, this leads to a gigantic number of states with many duplicates.

A near-duplicate detection algorithm can be a solution to limit the number of states found by Crawljax, while crawling the most essential, unique states. Through a literature survey it became apparent that Simhash and Broder are two state-of-the-art near-duplicate detection algorithms that are suitable for Crawljax.

In this project, both algorithms are implemented into Crawljax. These algorithms have been tested extensively to determine the performance of the new duplicate detection algorithms on Crawljax in comparison with the current version of duplicate detection. The testing has been done using a separate calibration tool, which can distribute tasks over different machines to lower the amount of time needed for the tests. This calibration tool will return the number of mistakes of every near-duplicate detection algorithms for many different parameter values. This make it possible to compare the performance of the near-duplicate detection algorithms.

The results of the calibration tool showed us that Crawlhash was faster, but Broder was slightly better.

Additionally the so-called threshold-slider has been designed to simulate what would have happened with the state-flow-graph of a crawl if a higher threshold was used. This makes it possible to find a nice threshold for one specific web application.

## 3 Introduction

A crawler is a program that automatically collects Web pages to create a local index and/or a local collection of web pages [3]. Usually, this is done by providing the crawler with a set of URLs, called seeds. The crawler will visit the websites of those URLs and search for new URLs by following the hyperlinks. In this way, the assumption can be made that every webpage has a unique URL. However this is not always true, because JavaScript and dynamic DOM manipulation on the client side of Web applications is becoming a widespread approach for achieving rich interactivity and responsiveness in modern Web applications [8]. With the techniques, known as AJAX, it is possible that several different states have the same URL.

Crawljax is specifically designed to solve this problem. Crawljax can explore any JavaScript-based AJAX web application through an event-driven dynamic crawling engine[1]. During the crawl, Crawljax will make a state-flow graph of the dynamic DOM states and the event-based transitions between them [6]. Crawljax was originally designed to make the dynamic content of websites visible to search engines. Later it was extended for automatically testing interactive web applications [7].

Naturally with crawling rich interactive Web applications some interaction between the user and the Web application leads to a new unique state and other interaction does not. The major problem with this is how to differentiate unique states from duplicate states. If every change leads to a new, unique, state the number of states will grow exponentially. Having too many of these duplicate or redundant states will lead to increases in crawl time and adds redundant states to the results. This is a problem for nearly all uses of Crawljax. It is therefore necessary that a form of near-duplicate detection is used to determine if a newly found state is a new unique state or a duplicate or redundant state.

In this report we will first identify the actual problem of this project in section 4 and 5. In section 6 all constraints and requirements of a possible solution to the problem are defined. The project methodology and planning will be explained to show how the general process went. After that, the general design of the important algorithms and classes will be explained in section 8. Afterwards in section 9, the implementation of this design will be discussed. Finally refsec:results will show the test-results of the algorithms.

## 4 Problem Definition

A crawler such as Crawljax attempts to find all possible states of a web application. A crawl is called an exhaustive crawl if the crawler has found all the states of the crawled web application. Exhausive crawling can lead to a gigantic number of states because each minor change in the DOM will result is a new state. This can result in the 'state explosion problem', where the number of crawled states grows exponentially in the number of pages on a website [8]. Adding one page to a website can result in an addition of a lot of states in the state-flow-graph. For example the new added page can have some counters, timestamps

[1]http://crawljax.com/about/

or advertisements, which can be different for every visit of the page. This way, every link to this new page adds a new state to the state-flow-graph, even though the relevant content has not changed. See appendix C for the original project description.

A demonstration of this problem is when a website contains a counter to show the number of visits to a certain page. Each time the crawler loads the URL, the counter will be incremented. This affects the DOM, which leads to a new state. The result is an infinite number of states, which leads to a crawl that is never exhaustive. Crawling too many states will slow down Crawljax. Secondly, the result will be of less use, because the resulting state-flow graph is cluttered with near-duplicate states.

The problem this thesis attempts to solve is: Let Crawljax recognise when it found a state that is already crawled, to make it possible for Crawljax to crawl all unique states of a web application without crawling duplicate states.

# 5 Problem Analysis

The solution to the problem just defined lies in determining when a change in the DOM does not warrant creating a new state. Our way of solving this problem in this project is to make use of a near-duplicate detection algorithm. This will make it possible to crawl a web application exhaustive in a reasonable amount of time, because the crawler finds only the unique states. When a near-duplicate state is found, the state should not be added to the state-machine and all its outgoing links to other states should not be crawled at that time. However, to determine whether a state is a near-duplicate of another state is not a trivial task. The first step is to get the relevant content of the state, which, in this case, is the main content of the state. This is due to the fact that we use the convention that two states are near-duplicates if the textual content on both states are almost the same. Using this definition all HTML-attributes, header-information and the white-space should be ignored, when comparing states. The last and most important step is to find out if the content is almost similar to another content. If both steps can be executed effectively, than it is possible to lower the number of states that a crawler has to crawl substantially.

**Current Situation**

Crawljax has a configuration that limits the number of states that will be crawled at maximum depth. The problem with this is that the crawl will stop the branch which reaches the maximum depth. But it can be possible that many different unique states exists on a higher depth. In this way, Crawljax can miss a lot of states when the maximum depth is to low. Making the maximum depth higher to find more states can result in way to many states, while crawling a lot of duplicates. This can make the runtime of the crawl very high. To solve this problem, Crawljax makes use of a duplicate detection algorithm based on a manipulation of the DOM. The manipulation of the DOM is called the stripped DOM. The stripped DOM contains only the HTML-attributes. Two states are duplicates if the stripped DOM of both states are exactly the same. According to this algorithm, two states are duplicates of each other if the stripped DOM of both states is exactly the same.

This approach provides decent duplicate-detection, for cases such as changes in timers.

Nevertheless there still many situations in which states that are duplicates in terms of content are seen as unique states by Crawljax. Such a situation is for example the DOM

```
<HTML><HEAD></HEAD><BODY><div></div></BODY>/HTML>
```

It is still possible that a $< p >< /p >$ element with some text will be added to the DOM, with a JavaScript button. So this can also lead to an infinite number of states, because every time Crawljax clicks on the button, the stripped DOM will be different.

On the other hand, the following situation could also occur. Suppose that the DOM

```
<HTML><HEAD></HEAD><BODY><p>...</p></BODY>/HTML>
```

has all of its textual content inside the $< p >$-element. When a user presses a button the $< p >$-element is filled with completely new, unique content. Using the current duplicate-detection algorithm, these states will all be regarded as one state. This is due to the fact that the algorithm only checks for changes in the DOM and completely disregards the textual content.

Both previous examples show that the current duplicate-detection algorithm in Crawljax is far from optimal. Occasionally it fails to detect duplicate states and there are situations were unique states are seen as duplicates. In this project we will try to solve the question: How to detect near-duplicate states in terms of the content?

# 6 Requirements

With the problem of the project defined, the constraints of a possible solution should be established. We define the global requirements in section 6.1. These will be expanded and divided into functional and non-functional requirements, in section 6.2 and 6.3 respectively. In section 6.4 a calibration tool will be introduced to find sensibe default values for the configuration-setting of a near-duplicate detection algorithm.

## 6.1 Global Requirements

To provide an overview of what the final product must be capable of, first three global requirements for the system that must be fulfilled are provided. By first painting a broader picture of the goals of the project, the more specific requirements are put into context.

- The product should provide better duplicate-detection than the current duplicate-detection implementation in Crawljax.
- The product should be seamlessly integrated in Crawljax, it should be highly configurable by the users.
- A blog post should be written about the product.

To expand on the global requirements provided above, we differentiate between functional and non-functional requirements, in section 6.2 and 6.3 respectively.

## 6.2 Functional Requirements

The functional requirements of the near-duplicate detection describes the components and functionality that defines what it is supposed to accomplish.

- Provided two states, the new near-duplicate detection should return a Boolean based on whether those two states are near-duplicates, and if possible also the distance between the two states.
- The product should be seamlessly integrated in Crawljax. It should be highly configurable by the users.
- The strictness of near-duplicates, should be noted by and adjusted by a threshold-variable.
- The user should be able to see implications of different thresholds by moving a slider in the Crawl-overview
- All configuration-settings of the new near-duplicate detection should have the option to be adjusted by the user in the Crawljax-Configuration.

## 6.3 Non-functional Requirements

During the initial meetings, the importance of high code quality was stressed by the client. Therefore, several non-functional requirements were identified to which the project should adhere to.

**Efficiency**
The new near-duplicate-detection should be able to process the states in a reasonable time-frame, preferably not slowing down Crawljax.

**Maintainability**
Maintainability is a highly important aspect of the code, as stressed by the client. The code should be of at least the same level of quality as the existing codebase. Additionally, the code should also be written in the same style as the existing code.

**Extendability**
As mentioned in the functional requirements, the users should be able to adjust all settings related to the near-duplicate-detection process. Additionally the code should be easily extendable for users wanting to make larger changes to the implementation.
Besides the extendable nature of the code, the code should also be as backwards-compatible as possible. Users should only need to make minimal changes to their existing code to run the new version of Crawljax.

**Reliability**

The new near-duplicate-detection should outperform the old implementation. This implies that the new implementation should provide better results on average than the original implementation. Additionally, the new implementation should provide stable results, when repeating crawls.

**Usability**
Preferablly, no additional (hardware) dependencies should be introduced. The new version of Crawljax should support and run on the same platforms as previous versions of Crawljax.

### 6.4 Calibration Tool Requirements

The near-duplicate detection has some configuration-settings, as described in the functional requirements 6.2. We need to define sensible default values for these configurations. We will set up a calibration tool to collect these values. The tool will test the algorithm on as many different values and combinations as possible. This calibration tool will exist alongside Crawljax, but will not be integrated into the Crawljax-project. Therefore the initial requirements of the calibration tole won't need to be as strict as the Crawljax-code.

- The calibration tool should automate the task of testing and comparing the different settings of the near-duplicate-detection algorithms.
- It should allow for distribution of website-crawls over multiple computers, to decrease the workload on a single computer.
- The suite should provide the user with some objective metrics about the performance of the current setup of the near-duplicate-detection.

With this calibration tool, we can also measure the efficiency and reliability of a new implemented near-duplicate detection algorithm. This can be done by comparing the runtime and the number of mistakes of the current version and a new version of the near-duplicate detection.

# 7 Project Methodology

In this section the methodology of the project is discussed. Additionally, the planning and the tools that were used throughout the process.

## 7.1 Process Strategy

The entire project from the initial research up to developing the systems took place at the Delft University of Technology (TU Delft), because Crawljax is a open-source project mainly maintained by the TU Delft. Working in a team of two persons results in a different strategy compared to the strategies used by larger teams. Such a small team allows for working for constant communication and discussion. Such a situation are very suitable for agile development processes.

**Test-Driven Development**
Testing and code quality are two key aspects in this project. Therefore, test-driven development (TDD) will be applied in this project. This approach was particularly useful due to the explorative nature of the project. By writing the tests first, the, sometimes difficult, algorithms can be correctly implemented based on the test cases.

**Agile Programming**
Besides the Test-Driven Development approach a iterative approach fitted this project well. Due to the explorative nature of the project a strict long-term planning would not work. At the start of the project there was no real indicator of what issues would come up throughout the process. Besides, the team existed out of just two students. Small teams are ideal for Agile Programming, because of the constant communication.

## 7.2 Communication

For this project a high level of independent working was needed. The open-source nature of Crawljax also means that there is no company or managers, who actively monitor progress. On one hand, this allows a lot more freedom on all aspects. On the other hand, it is harder to keep the project on the right track. To know whether the direction the team takes is the right one. To prevent focusing on the wrong aspects, two procedures are implemented to improve communication.

**Meetings**
Especially at the beginning of the project, every week a meeting with client Alex Nederlof to discuss our progress helped a lot to differentiate main issues from the side issues. Additionally, we set up a Skype-meeting with the original author of Crawljax, Ali Mesbah, at the end of the project to discuss our product.

**Updates**
Besides the physical meetings, the team kept all interested parties informed about the progress and most important decisions. The informing was achieved by sending a email at the end of every week. In this email the progress of the week was reported, along with the planning of next week. Additionally, any major issues or decisions were explained, so all parties could review our decisions and, if necessary, adjust the course of the project.

## 7.3 Planning

As explained in the previous section, it was clear that a very flexible planning was required for this project. Therefore, the project only identified two phases, namely the research-phase and the prototyping/implementation-phase. The original planning called for a extensive research-phase of at least 2 weeks. See appendix B for the initial action plan. The summarized logging of the actual work is:

week 1 - 2:

- Decided on the delivarables, requirements and planning of the project with Alex and coach Arie van Deursen in the first meeting.
- Investigated the ins and outs of Crawljax.
- Started researching relevant literature.

Week 3:

- Build initial version of the testing suite, which is able to crawl all websites from a list with settings defined in a separate file.
- Finished up research report.

week 4 - 5:

- Implemented Crawlhash near-duplicate detection algorithm, see section 8.1.2.
- Tested performance of Crawlhash.
- Added distributed crawling functionality to the testing suite.

week 6 - 7:

- Implemented Broder's near-duplicate detection algorithm, see section 8.1.3.
- Improved the structure of the Crawlhash to make switching to Broder easy.
- Tested performance of Crawlhash and Broder.
- Added analysis-functionality to the testing suite.

week 8 - 9:

- Implemented the threshold-slider in the Crawloverview-plugin, see section 8.3.
- Major code cleanup and sent the code to SIG.
- Testing out configurations of the near-duplicate detection algorithms to find optimal values.
- Feedback of SIG and Alex incorporated into the project.

week 10:

- Code improvements based on static code analysis tools
- Refactoring of code to make it easier for users to change the near-duplicate detection.
- Additional testing out configurations of the near-duplicate detection algorithms to find optimal values.

week 11 - 12:

- Added an example to Crawljax-core.
- Slight changes to the code.
- Write the final report according to the requirements on Blackboard.
- Prepare final presentation.
- Send code to SIG.
- Integrate code into official Crawljax

## 7.4 Tools

Throughout the project many different open source tools were used that improved the development process in general and the quality of the software that was being developed. In this section a short description is given for each of the tools.

**Git and Github**
At this moment in time Github[2] is extremely popular in the open-source community. Crawljax is no exception and is also located on Github. As the emphasis of this project is on open-source development using Git[3] as version-control system, the choice of using Git was not difficult.

**Maven**
Maven[4] is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

**JUnit and Eclemma**
The keyword of this project is testing. Therefore, automated testing is essential. JUnit[5] is the standard in Java for automated unit tests. Additionally, Eclemma[6] shows the total

[2]http://github.com
[3]http://www.git-scm.com
[4]http://maven.apache.org/
[5]http://junit.org/
[6]http://www.eclemma.org/

code coverage. This makes sure that no methods remain untested.

**Eclipse**
With all of the tools mentioned, a IDE with support of all these tools would be incredibly convenient. Eclipse[7] is a mature open-source IDE for primarily Java with plugins for all these tools.

# 8 Design

This project consist of two separate tasks. The first one is to develop a near-duplicate detection algorithm for Crawljax. The design for two state-of-the-art near-duplicate detection algorithms is shown in section 8.1. The second task is to make a calibration tool to find good parameter values for the near-duplicate detection algorithms and measure its performance. The design for the calibration tool is given in section 8.2. During the project, the client mentioned that it would be nice to build a 'threshold-slider' in the crawloverview of Crawljax. This crawloverview is generated by Crawljax after a crawl of a web application. A threshold in one of the parameter values of the near-duplicate detection algorithms. The threshold determined the maximum level of the difference between two states to mark the states as duplicates. The slider should make it possible to change the threshold for a near-duplicate detection algorithm after a website is crawled. For more information see section 8.3.

## 8.1 Duplicate detection

The 'state explosion problem' where the number of crawled states grows exponential with the number of pages and events on a website is discussed in section 4. We will try to solve this problem with state-of-art near-duplicate detection algorithms. For Crawljax we will use two state-of-the-art near-duplicate detection algorithms: Crawlhash, also known as Simhash, and Broder's algorithm. See *appendix A* for the introduction and explanation of these algorithms from our research-phase in the first weeks of this project. In this section we will describe these algorithms with a running example. In section 8.1.1 we will start the example with generating features from some content, which will be used for calculating the difference-distance between two states by Crawlhash and Broder in section 8.1.2 and 8.1.3 respectively.

### 8.1.1 Features

An important aspect of all near-duplicate-detection algorithms is the selection and weight-assignment of features. Features are low dimensional mappings of high dimensional properties of the document/state. Nearly every commonly used duplication-detection research, including Broder et al. [1] and Charikar [2], explains their algorithms using words. They

[7]http://www.eclipse.org/

use a set of unique *w*-shingles to split the text of a document into features. A shingle is a subsequence, which can contain for example a fixed number of characters, words or sentences. Both implemented algorithms, Simhash and Broder, can use shingles as feature-type. Shingles originates from the analogy with roof tiles, as roof tiles tend to overlap each other. To make this more clear, we will show an example. For this example we take words as the feature-type-size and 3 as the feature-size. This means that a shingle, in this context, is a sequence of 3 continuous words. Imagine that a state has the following content:

"Hello friends! This is my personal website."

This content we be divided into 3-word-shingles as follows, use algorithm 1 with n=3:

**Algorithm 1** Divide document into n-word-shingles

```
Divide the content into words
while it is possible to take n words do
  Take the first three words
  Remove the first word
end while
```

The first step gives an array of words:

{"Hello", "friends!", "This", "is", "my", "personal", "website."}

The second step creates the features:

{"Hellofriends!This", "friends!Thisis", "Thisismy", "ismypersonal", "mypersonalwebsite."}.

The elements in this list of the second step are called features, which together represent the content of the state. The state has 7 words of content and we used 3 words as the shingle size. This means that the number of obtained shingles will be equal to 7 - (3 - 1) = 5, or more abstract: number of elements in the content of the state - (number of feature-size - 1).

This statement holds because for each new word in the content, a new shingle will be added. However, there will not exist a single shingle if there are only one or two words in the state, so we have to subtract this number from the total. This gives us 7 - 2 = 5.

### 8.1.2 Simhash/Crawlhash

Currently Charikar's Simhash algorithm is one of the most popular near-duplicate detection algorithms [4]. The Simhash algorithm [2] is a dimensionality reduction technique. It maps high-dimensional vectors to small-sized fingerprints [5]. In the research report (Appendix A) the basic concept and the algorithm itself is explained. To demonstrate this algorithm, an example will be described in section 8.1.1.

**Generate fingerprint**
Each states has its own fingerprint, which can be used to compare the similarity between two states. The fingerprint is basically a hash with the property that relatively similar values will be hashed to relatively similar hashes. The word relatively is important. It is trivial that exactly the same values will be hashed to the same hash, but it is not trivial that almost similar values will be hashed to almost similar hashes. This is different from the usually used hashes, such as SHA256, because they will generate a completely different hash when the value changes, even when the change is very small relative to the size of the value.

**Continue the shingle example**
Consider the example from section 8.1.1. The generated featers are:

{"Hellofriends!This", "friends!Thisis", "Thisismy", "ismypersonal", "mypersonalwebsite."}.

The first task is to hash all the features with a simple hashing algorithm. The result of the hashCode() method in java will give the following result:

{01101101111010011110001110011100, 00001110001011110111000010010100, 01001101000111011010110000010100, 00111100101000001110100010010110, 11110010101010010100111011011111}.

All these hashes will be used to create the fingerprint of the content of the state. To make this clear, we will put the hashes under each other and calculate the fingerprint. The values in the field 'Addition' are calculated by the values above. Add one if the value is 1 and subtract one if the value is 0. Finally, the fingerprint bit is 1 if the value in the row 'Addition' is greater that 0 and the fingerprint bit is 0 if the value in the row 'Addition' is less or equal than 0.

| Bit number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| Addition | -3 | 1 | 1 | -1 | 3 | 3 | -1 | -1 | 1 | -3 | 3 | -3 | 3 | -1 | -3 | 3 |
| Fingerprint | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

| Bit number | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| Add | 1 | 3 | 3 | -3 | 1 | -1 | -1 | -3 | 3 | -3 | -5 | 5 | -1 | 5 | -1 | -3 |
| Fingerprint | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

This will give the fingerprint '01101100101010011110100010010100' of the value 'Hello friends! This is my personal website.' using a shingle-size of 3-words.

**Small change in the content**

Lets change the content of the example a bit to see how the fingerprint changes. The new content will be:

'Hello world! This is my personal website.'

Features of the content:

{"Helloworld!This", "world!Thisis", "Thisismy", "ismypersonal", "mypersonalwebsite."}.

Hashes of the features:

{11100111011011001000110000111111, 01110011001101011101011111110111, 01001101000111011010110000010100, 00111100101000001110100010010110, 11110010101010010100111011011111}

This will change two of the five hashes. These rows are indicates with arrows.

| Bit number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| → | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| → | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| Addition | -1 | 3 | 3 | 1 | -1 | 1 | -1 | 1 | -1 | -3 | 3 | -1 | 1 | -1 | -5 | 1 |
| Fingerprint | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

| Bit number | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| → | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| → | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| Add | 3 | 1 | -1 | -3 | 3 | 3 | -1 | -3 | 1 | -1 | -1 | 5 | -1 | 5 | 3 | 1 |
| Fingerprint | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

So the fingerprint of

’Hello world! This is my personal website.’
=
’01110101001010011100110010010111’

**Getting the similarity distance between states**
In Crawlhash, the fingerprints can be compared using the Hamming distance. A small Hamming distance indicates more similarity between two documents. In the previous example, the change of the word ’friends’ to ’world’ caused the hash to change 8 of the 32 bits.

011**10**10**10**010100111**0**01**1**00100101**11** ← Fingerprint of the content with the word ’world’.
011**01**10**01**010100111**1**01**0**00100101**00** ← Fingerprint of the content with the word ’friends’.

**Advantages of crawlhash**
Crawlhash creates a small-sized fingerprint of the content of a state, which makes it easy to calculate the distance between two states by just finding the hamming-distance of the two fingerprints. This can be done in a constant time and space. The space is of complexity O(1) because each state needs only 32-bits to save the fingerprint. The time is of complexity O(1), because only the 32-bits of two fingerprints should be compared to each other for differences.

**Disadvantages of crawlhash**
Crawlhash has some level of randomness. Every feature with different content will be hashed to a different hash; one can see this hash as a random sequence of zeros and ones. Because of this, it is not possible to predict the exact number of differences in the fingerprints when there is a change in the content. It is only possible to predict some bounds or make a confidence interval. Assume we use one-word-shingles for the content ”Nothing or Broder”. When we change this to ”Crawlhash or broder”, it will probably give another result as when we had changed the content to ”Something or broder”. This is not intuitive, because we use one word to define the features and in both cases only one word changes. It would be nice if the the number of different bits in the fingerprints would be the same. The difference occurs because the hash of ’Nothing’ can have some bits in common with ”Crawlhash” but not with ”Something” or otherwise. This can result in a different value for a bit in the fingerprint when both bits of the other two hashes are different. This due to two of the three hashes that will remain the same. ”or” and ”Broder” remain unchanged. So for the bit-numbers where the bits of both hashes are the same (both 0 or 1). The possible change in the bit of the third hash will not affect the fingerprint. This problem disappears largely when there are a lot of features and just a few that are different. Thus to get rid of this disadvantage, more and smaller feature-sizes should be preferred above less and bigger feature-sizes.

### 8.1.3 Broder's algorithm

Another way of finding duplicates is to use the number of common shingles in comparison with the total number of different shingles. Broder's duplicate detection [1] uses features to divide the content of a state into a set of hashes These hashes will be used to compute the resemblance with another state by calculating the Jacquard-coefficient. In this case, similar to Crawlhash, shingles were used to generate the features.

**Jacquard-coefficient**
The set of features will be hashed and saved for each state. This set of hashed features is used to compare the resemblance of two documents by calculating the ratio of the intersection divided by the union of the elements of the sets. This ratio is called the Jaccard coëfficient: $\frac{|S(d) \cap S(c)|}{|S(d) \cup S(c)|}$, where $d$ and $c$ are documents, $S(d)$ and $S(c)$ are the the sets of hashed features and $|S(d)|$ is the number of elements in the set $S(d)$. By using a threshold $t \in (0,1)$, we can say that two pages are near-duplicates of each other when the ratio is higher that $t$.

**Consider again the example content**

d = "Hello friends! This is my personal website."

As shown, this give the set of features

{"Hellofriends!This", "friends!Thisis", "Thisismy", "ismypersonal", "mypersonalwebsite."}.

The next step is to hash these features. Basically, every feature that is not exactly the same is hashed to a different hash. The make this example more clear, we will just use the string-representation of the hashes instead of the hashes.

Now make a small change in the content

c = "Hello world! This is my personal website."

This gives the features

{"Helloworld!This", "world!Thisis", "Thisismy", "ismypersonal", "mypersonalwebsite."}.

**Calculate the Jacquard-coefficient**
The intersection of S(d) and S(c):

{"Thisismy", "ismypersonal", "mypersonalwebsite."}

The union of S(d) and S(c):

{"Hellofriends!This", "friends!Thisis", "Thisismy", "ismypersonal", "mypersonalwebsite.", "Helloworld!This", "world!Thisis"}

This will makes the Jacquard-coefficient equal to : $3/7 \approx 0.43$
The value would be 1 if both states are completely the same and 0 if they are both completely different. A value in between represents the corresponding similarity.

**Advantages of Broder**
Broder's algorithm has less issues of randomness as crawlhash, see 'Disadvantages of Crawlhash' in section 8.1.2. "Just", "a","flesh" and "wound" will be all hashed to different hashes. Each sequence of two states with content "Just or a", "a or flesh" and "flesh or wound" will give a total of 2 elements in the intersection and a total of 4 elements in the union. So the similarity-value will be $2/4 = 0.5$. This value say that these three states will be near-duplicate if the threshold is at least 0.5. The advantage of Broder's algorithm is that the results will be more stable and reliable.

**Disadvantages of Broder**
The calculation of the difference-distance takes more time and space than the algorithm of crawlhash. Space has a complexity of O(n), because content of n words/chars/sentences has n-p+1 features, where p is the feature-size. All the hashes of these features should be saved. Time has a complexity of O($n^2$): Union and the intersection can be in O($n^2$)-time. Every element in first set should be compared with every element in the second set in the worst-case scenario. The count of the elements in the sets can be done in linear-time and the division O($n^2$). Therefore, the total complexity of time is O($n^2$). So compared to Crawlhash, Broder takes in more memory and takes more computation time.

## 8.2 The testing environment

There are several ways to find duplicate or near-duplicate states during a crawl, see section 8.1.2 and 8.1.3. These different near-duplicate detection algorithms should be compared with each other to decide which algorithm and which parameters will perform the best for Crawljax. The comparison should give results, which make it possible find a reliable near-duplicate detection algorithm for Crawljax.

In order to test the different near-duplicate detection algorithms, a automated calibration tool is needed. It was expected that several hundreds of test runs would be needed to test a major number of different parameter situation. Doing the comparisons manually would cost far too much time and the comparisons would be prone to human error. Therefore it was concluded that a automated testing tool was needed.

This calibration tool would be responsible for three aspects of the comparisons. First of all, the tool should be able to run Crawljax-crawls with configurable settings. This way we can easily test out various types of the near-duplicate detection algorithms by changing the parameters. Secondly, the testing suite should be able to distribute the websites to be crawled over multiple computers. By distributing the workload, the overall run-time of the tests can be reduced. Finally, the calibration tool should be able to compare the crawled results. These three aspects are represented in the Crawljax Calibration Tool as the three major components. The CrawlManager Component, the distributed component and the analysis component. These components will be covered in depth in sections 8.2.1, 8.2.2 and 8.2.3 resplectively. Finally, in section 8.2.4 we will explain how the the optimal result of a website is found, which is necessary for the analysis component.

### 8.2.1 CrawlManager Component

The most essential component of the testing suite is the link between the testing suite and the Crawljax-core. This component is responsible for mapping the configurations to a Crawljax-configuration and launching Crawljax-sessions. As the name already suggests, the ConfigurationMapper is the class that will be responsible for mapping the configurations set in the testing suite to a configuration compatible with Crawljax. The CrawlManager class is responsible for properly setting up and launching a Crawljax-session. Additionally, to make the testing suite already fully functional, this component is able to read websites and settings from local files. It will use the settings from the file to configure the Crawljax-crawler and it will keep crawling until all websites from the file have been crawled.

### 8.2.2 Distributed component

The whole idea of having a separate calibration tool is to be able to distribute the crawling of a large collection of websites over multiple computers. This is where the distributed component comes into play. The distributed component is responsible for distributing tasks over a number of computers. A task will be the crawl of a certain web application. To distribute these tasks require two types of actors. At least one computer or server, the distributor, should be in charge of distributing the tasks between the other computers, called the workers. These workers just perform some tasks, i.e. crawling some websites and return the output to the server. This configuration is known as a master-slave configuration, where the workers act as slaves.

The main requirements for the distributed component in this situation, would be that these workers should be flexible, should be easy to setup and should have the least necessary interaction with (human) supervisors. Additionally, the workers should not impact the entire session, when they fail to crawl a website.

There are three separate types of data which need to be distributed. These are the tasks for the workers, the configuration-settings and the results of the crawls. Each of these types of data has been implemented as a separate sub-component.

Instead of a traditional master-slave configuration, we opted for a slightly adjusted configuration. The distributor is just a database-server, accessible over the network. When workers are idle, they will poll the database to check if there are any free tasks available. If there is a free task available, the worker will claim the task by updating the specific row in the database. Other workers recognize the task as being claimed or not free.

After the worker has claimed a task, it will retrieve the relevant set of configurations. Each task can have custom and/or common settings associated with it. The CrawlManager will be called to setup the actual crawling. Afterwards the results of the crawl will be sent to the database using the ResultProcessor. Finally to indicate that the task has been

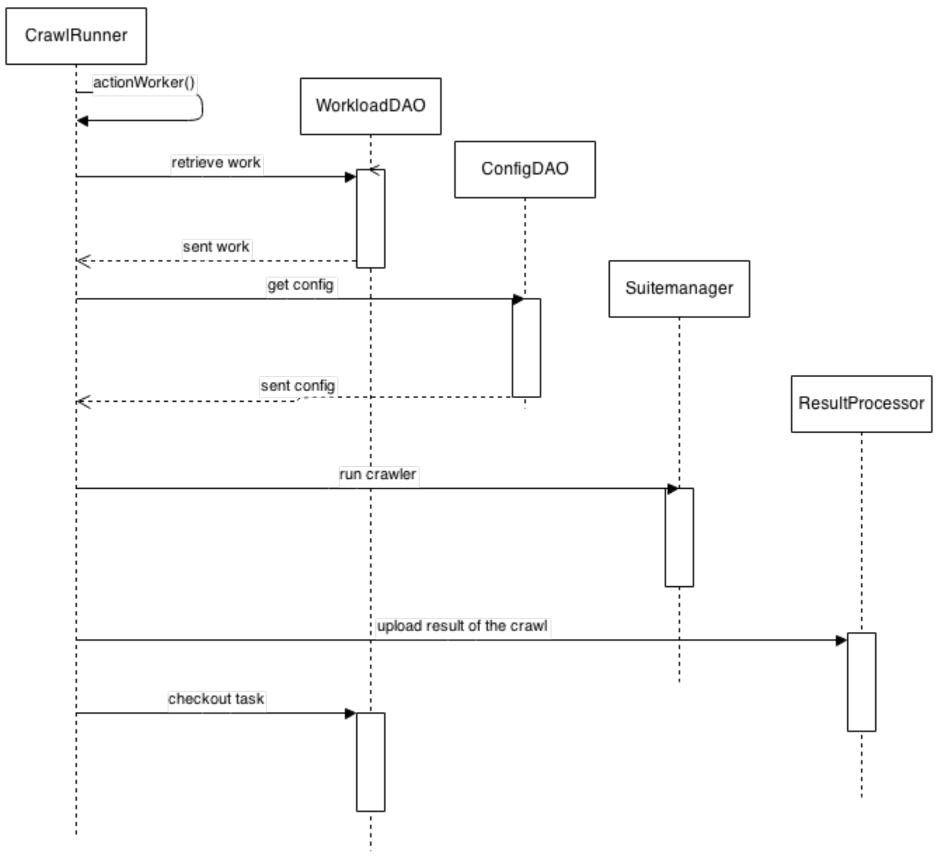


Figure 1: Sequence diagram of a worker/slave that performs a task

successfully completed the task is marked as completed in the database. The whole path that a worker performs is shown in the sequence diagram in Figure 1.

### 8.2.3 Analysis Component

The component that makes this software a calibration tool is the analysis component. The analysis component can be used regardless of whether the websites have been crawled locally or using the distributed component. It is responsible for processing and comparing the crawled results.

The analysis component measures the effectiveness of the near-duplicate detection algorithms, which are described in section 8.1. There should be a set of websites, for which all the duplicate states have been manually marked, see section 8.2.4 how this will be done. The analysis component compares a crawled set with a specific near-duplicate detection algorithm with the perfect/benchmark set. The level of similarity between the crawled set and the benchmark set provides an indication on the effectiveness of the used near-duplicate detection algorithm.

The component has been designed to be very flexible and extendable, to easily allow changes in the way the crawls are analysed without having to change several parts in the software. The component can be subdivided into three sub-components, namely metrics, processors and the analysis-core.

**Analysis-core**
The analysis-core is the only vital sub-component. It contains the AnalysisBuilder, Analysis and the Statistic-classes. The Analysis-class is at the center of the analysis component. An analysis-object represents a single comparison between the benchmark-set and the newly crawled set. If the analyse is completed it also contains the statistics of the comparison. The AnalysisBuilder is responsible for fetching the benchmark-set and the crawl or retrieve the new set. In our case the AnalysisBuilder will fetch the benchmarked-set from the database and will issue a new distributed crawl for the new set. Finally there is the Statistics-data object. This object is responsible for holding a single statistic about an Analysis. Without any metrics the set of Statistics in an Analysis will be empty.

The set of metrics in a comparison are responsible for the way the comparison between the benchmark-set and the newly crawled set is carried out. Each metric compares a single attribute between the two sets. It reports the results of the comparison by returning a Statistic-object. Metrics can easily be created and added to the analysis by having a class implement the IMetric-interface and adding it to the AnalysisBuilder.

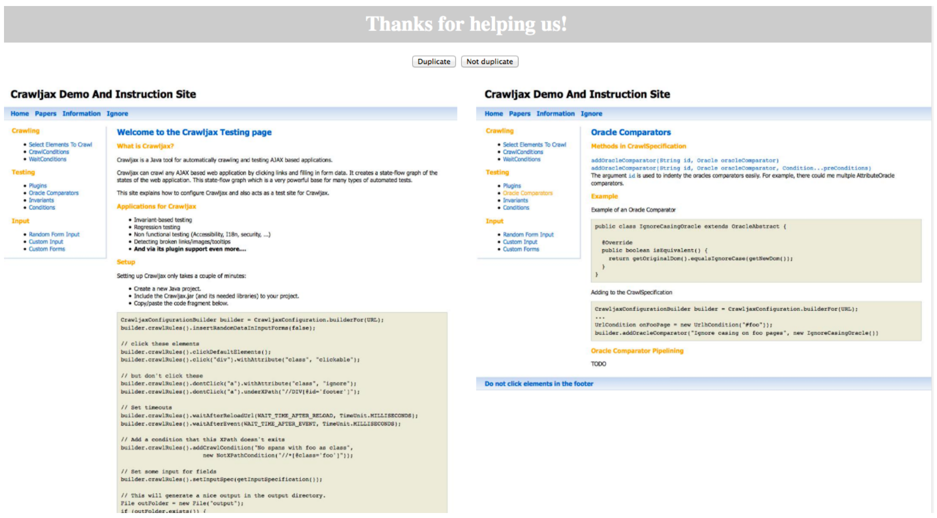
**SpeedMetric**
The speedmetric will compare the runtime of the test with the runtime of the crawl we used to find the optimal result. The result of the metric will be the percentage of the increase in runtime. The runtime of the crawl with the new duplicate-detection will be lower/faster if the percentage if is positeve and higher/slower if the percentage is negative. The percentages of different crawls with different duplicate-detection can be compared with each oter to see the change in runtime.

**State-analysis metric**
This metric compares the states of the two results. It will identify missed states and duplicate states. Missed states are states that the crawl of the test has not included in the resulting set, whereas the optimal/benchmark set do have these states. Duplicate states are the states that are in the result, but are not in the optimal/benchmark set. In this definition, missed states are false positives (the duplicate-detection says that the states are duplicates, but they're actually different) and duplicate states are false negatives (duplicate-detection says that the states are not duplicate, but they're actually are duplicates).

**Processors**
After the comparison is completed, processors are responsible for processing and/or storing the Analysis-object data. Multiple processors can be run on the same analyse-data independently of each other. The only requirement is that a processor extends the AnalysisProcessor-interface.

### 8.2.4 Annotation tool/website

The analysis component, described in section 8.2.3, needs benchmark sites to compare the result of a crawl with. We obtain this benchmark site through crawling all states of the site. This is done by setting the crawl-depth on infinite and using a duplicate detection on the whole DOM, where every change results in a new state. Once all these states are found, all the duplicate states should be found manually to be sure that the result will be optimal. To find the duplicates in terms of content, we made a website to make this process faster and to lower the possibility of mistakes. The website shows the screenshots of every possible combination of states. For each combination, the user needs to tell the website if the combination is a duplicate or not. Figure 2 shows a screenshot of this website. The definition we used to decide when a state is a duplicate of another state is defined in definition 1. The two states are sent to the database as a tuple if the combination is marked as duplicate. After the decision is made, the next combination will be shown. This process will be repeated until the last combination is reached.



Welcome to the Crawljax Testing page

What is Crawljax?

Crawljax is a Java tool for automatically crawling and testing AJAX based applications.

Crawljax can crawl any AJAX based web application by clicking links and filling in form data. It creates a state-flow graph of the states of the web application. This state-flow graph which is a very powerful base for many types of automated tests.

This site explains how to configure Crawljax and also acts as a test site for Crawljax.

Applications for Crawljax

- Invariant-based testing
- Regression testing
- Non functional testing (Accessibility, I18n, security, ...)
- Detecting broken links/images/tooltips
- **And via its plugin support even more....**

Setup

Setting up Crawljax only takes a couple of minutes:

- Create a new Java project.
- Include the Crawljax.jar (and its needed libraries) to your project.
- Copy/paste the code fragment below.

```
CrawljaxConfigurationBuilder builder = CrawljaxConfiguration.builderFor(URL);
builder.crawlRules().insertRandomDataInInputForms(false);

// click these elements
builder.crawlRules().clickDefaultElements();
builder.crawlRules().click("div").withAttribute("class", "clickable");

// but don't click these
builder.crawlRules().dontClick("a").withAttribute("class", "ignore");
builder.crawlRules().dontClick("a").underXPath("//DIV[@id='footer']");

// Set timeouts
builder.crawlRules().waitAfterReloadUrl(WAIT_TIME_AFTER_RELOAD, TimeUnit.MILLISECONDS);
builder.crawlRules().waitAfterEvent(WAIT_TIME_AFTER_EVENT, TimeUnit.MILLISECONDS);

// Add a condition that this XPath doesn't exits
builder.crawlRules().addCrawlCondition("No spans with foo as class",
                            new NotXPathCondition("//*[@class='foo']"));

// Set some input for fields
builder.crawlRules().setInputSpec(getInputSpecification());

// This will generate a nice output in the output directory.
File outFolder = new File("output");
if (outFolder.exists()) {
```

Oracle Comparators

Methods in CrawlSpecification

`addOracleComparator(String id, Oracle oracleComparator)`
`addOracleComparator(String id, Oracle oracleComparator, Condition...preConditions)`
The argument `id` is used to indenty the oracles comparators easily. For example, there could me multple AttributeOracle comparators.

Example

Example of an Oracle Comparator

```
public class IgnoreCasingOracle extends OracleAbstract {

  @Override
  public boolean isEquivalent() {
    return getOriginalDom().equalsIgnoreCase(getNewDom());
  }
}
```

Adding to the CrawlSpecification

```
CrawljaxConfigurationBuilder builder = CrawljaxConfiguration.builderFor(URL);
...
UrlCondition onFooPage = new UrlhCondition("#foo"));
builder.addOracleComparator("Ignore casing on foo pages", new IgnoreCasingOracle())
```

Oracle Comparator Pipelining

TODO

Figure 2: Screenshot of the annotation tool

**Definition 1.** *State B is a duplicate of another state A iff all relevant content is the same in both states.*

People come to a site for certain information. We define the relevant content as the textual information on a site what for the user may visiting the site. All other things, like advertisement, timestamps, navigation lists can also be present on a site, but don't give the visitor the information where he visits the site for, usually.

## 8.3 Threshold-slider in the crawloverview plugin

For the near-duplicate detection we need some threshold to determine the 'near' in the word 'near-duplicate detection'. Two states that are not exactly the same, can be seen as duplicate or unique states. We will use a threshold to define the maximum allowed differences for which two states are duplicates. Two states are duplicate if the distance-value obtain by Crawlhash or Broder, see section 8.1.2 and 8.1.3, is lower than the threshold. Now that we know how we can determine It would be nice to change the threshold after the crawl to see what will happen when the value for the threshold changes. This will be done with the so-called threshold-slider, which will be introduced in section 8.3.1. The goal is to get the same result with the slider as with a new crawl with the threshold of the value defined by the slider. For example, crawl with threshold 2, move the slider to the threshold 7 and check if this will give the same result as a new crawl with the threshold 7. In section 8.3.3 we will see that it is not possible to guarantee that the results will be the same. Section 8.3.2 will show how the graph is build when the slider changes.

### 8.3.1 Changing the threshold with a slider

A slider will be placed on the website that is generated in the Crawloverview-plugin. Using this slider an user will be able to change the threshold and see how the graph will react. This makes it possible to perform only one crawl and see what would have happened when you have chosen a different value for the threshold. It is therefore not necessary to run Crawljax again on the same website with the new threshold value, to view the differences.

When Crawljax crawls a new web-page, the fingerprint of the new content will be checked against all fingerprints of all the states that are currently in the graph. If the fingerprint can be matched with one of the existing fingerprints, the new state is a duplicate. If this is the case, than the new state will not be added to the graph. So it is possible for Crawljax to run with a certain threshold and discard near-duplicate states 'on the fly', without saving all the duplicate-distances to the other states. This is not possible in the threshold-slider, because it would take too much time to run Crawljax again for each different threshold. To be able to determine duplicates with a different threshold without recrawling the website the duplicate-distance is saved of each state with all other states. This is done in the Crawloverview-plugin. This will make it possible to simulate what will happen to the graph if the threshold changes.

### 8.3.2 Slider algorithm

The threshold changes with a simple HTML-based slider. Upon a change every state will be compared to check if the state is still a unique state. If this is the case, than the algorithm will continue to the next state, otherwise the state will be removed from the graph. If a state is removed from the graph, a new state should not be compared against this already removed state. To give a more formal description of the algorithm:

**Algorithm 2** Callback of the threshold-slider

```
statesInGraph ← empty array
for all state in states do
    for all distance in state.duplicateDistance do
        if statesInGraph contains distance.otherState && distance $\leq$ threshold then
            remove state from the graph
        end if
        if state is not deleted then
            add state to statesInGraph
        end if
    end for
end for
removeUnconnectedStates(graph)
```

States, which should be sorted from old to new, can for example contain the following array:

```
"states" : [ "index", "state2", "state3", "state4",
             "state13", "state14", "state16", "state17" ]
```

And state4.duplicateDistance can be for example:

```
"state4" : {
      "duplicateDistance" : {
                 "index" : 0.8224163027656477,
                 "state2" : 0.8503649635036497,
                 "state3" : 0.8170914542728636
      }
}
```

If statesInGraph has [ ”index”, “state3” ] state4 will only be compared with Index and State3 to determine if state4 is a duplicate. State2 in duplicateDistance of state4 will be ignored. This is done in the first check in the if-statement: statesInGraph contains distance.otherState. The second check verifies if the distance between the two states i below the threshold: item.value <= threshold. If both conditions are true, the new state is a duplicate of a state that is already in the graph. So the state would be removed from the graph. One interesting, obvious, limitation to the algorithm is that compared to the initial threshold all other thresholds should be larger. This is due to fact that it is relatively easy to remove states from graph, which happens at larger threshold. On the other hand it is impossible to add states, so lower thresholds than the initial threshold are impossible to display correctly. Notice that the first time graph first be build according to the initial threshold. If the slider is used, states will be removed according to the value of the new threshold. It is possible that the only parent-node of node v will be deleted, but that node v will still exist. The result will be some different disconnected graphs. The method removeUnconnectedStates(graph) will solve this problem by remove all disconnected states of the graph. Just removing these nodes is a justified way of handling these nodes. A new crawl with that threshold will not crawl any links of the removed state, because the duplicate state will not be crawled.

### 8.3.3 Possible differences

As shown in previous sections, it is possible to get an indication of the different states that Crawljax will return if you change the threshold for the near-duplicate detection with the threshold-slider in the Crawloverview-plugin. Notice that the word 'indication' is used. It is not possible to get always the exact same result for a crawl with a threshold and changing the slider. Consider the following example:

- A crawl with a threshold 0.3 and change it with the slider to 0.6
- A crawl with threshold 0.6

These two result can be slightly different. The reason for this is that there can be states added with the crawl with a threshold of 0.6 that are not added with a threshold of 0.3. This seems strange, so here is an example:

Index is added.
State3: {
index -> 0.5
}
In this instance state3 will only be added in the crawl with a threshold 0.3 and not with the threshold 0.6

State5: {
index -> 0.7
State3-> 0.1 (Only for the crawl with threshold 0.3)
}

In this case, state5 will not be added for the threshold 0.3, because it is a duplicate with state3. In the crawl with a threshold 0.6, state 5 will be added, because it is not a duplicate of Index. It will not check if state5 is a duplicate of state3, as state3 is not present in the graph. It was already detected as a near-duplicate with Index.
One could say that state5 is near-duplicate of state3 and state3 is near-duplicate with Index, so state 5 is near-duplicate of index. This transitivity-attribute is true for exact duplicates, but not for near-duplicates. A readable example can be:
A = "My software never has bugs. It just develops random features."
B = "My software never has bugs. It constantly develops random features."
C = "My software never has bugs. It just develops cool features."

If the threshold for near-duplicate detection is one-word, two sentences are near-duplicates if only one or none words in the sentence are different. Then A is a near-duplicate of B, B is a near-duplicate of C, but A is not a near-duplicate of C, because they have two words different.

To conclude, near-duplicate detection does not have a transitive relation. The absence of this characteristic, can lead to slightly different results with the threshold-slider. The threshold slider is only meant as a nice tool for visualizing the general changes in the state-flow graph. It is not a replacement for actual recrawling a website with a different threshold.

# 9 Implementation

In section 8 the three tasks of the project are introduced: implement near-duplicate detection algorithm in Crawljax, make a calibration tool to test the performance of the near-duplicate detection algorithm on Crawljax and make a threshold-slider in the Crawloverview-plugin generated by Crawljax. Now the implementation details of these tasks will be discussed in section 9.1, 9.2 and 9.3 respectively.

## 9.1 Duplicate detection

There are important methods in the implementation of the near-duplicate detection algorithms that we will describe in this section. The class diagram of the duplicate-detection package we developed for Crawljax will be shown in section 9.1.1. The Hamming distance method and the Jacquard-coefficient method are the most notable methods which return the level of similarity between two states. Due to this importance, the implementation details of these methods will be explained in section 9.1.2 and 9.1.3 respectively.

### 9.1.1 Class diagram

Figure 3 shows the class diagram of the duplicate-detection package that was developed for Crawljax. In the interface, *DuplicateDetection* and *Fingerprint* are the central part of this package. The first interface must be implemented to add a near-duplicate detection algorithm, Crawlhash and Broder. These algorithms generate objects of the interface *Fingerprint*. *Fingerprint* is the basic element generated by a *DuplicateDetection*, which essentially an abstraction of the underlying hashes and comparison methods. In *Fingerprint*, the method getDistance is defined. This method return a value that represents the distance between two states. This means that the distance is low for similar states and high for different states. The implementation of Broder DuplicateDetection uses the Jacquard-coefficient to determine the distance, while Crawlhash uses the Hamming distance. Notable is that the Jacquard-coefficient gives the value 1 (high) if two states are completely the same and 0 (low) if the states are completely different (both states does not have nothing in common). With the Hamming distance the values are inverted, where high values indicate no similarity. To make the getDistance-method more consistent the result of the Jacquard-coefficient has been inverted. This is done by one minus the result of the Jacquard-coefficient. This will give us a value between the 0 and 1 that represent the level of difference between the states, where high values indicate more differences between the states.

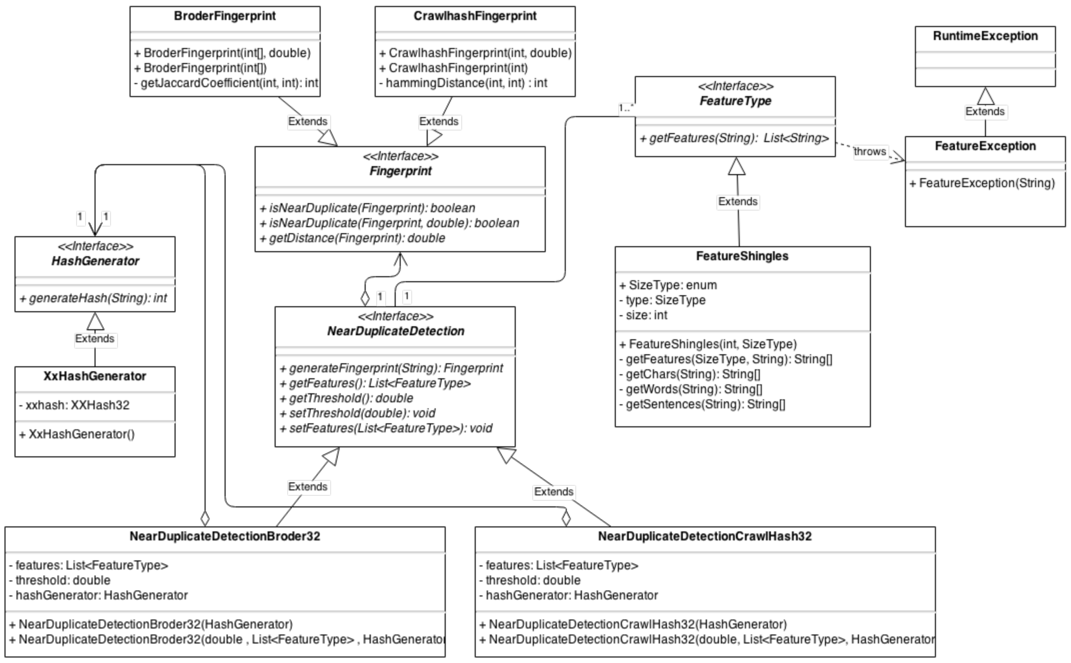

BroderFingerprint
+ BroderFingerprint(int[], double)
+ BroderFingerprint(int[])
- getJaccardCoefficient(int, int): int
CrawlhashFingerprint
+ CrawlhashFingerprint(int, double)
+ CrawlhashFingerprint(int)
- hammingDistance(int, int) : int
RuntimeException
<<Interface>>
FeatureType
+ getFeatures(String): List<String>
1..
Extends
Extends
Extends
throws
FeatureException
+ FeatureException(String)
<<Interface>>
Fingerprint
+ isNearDuplicate(Fingerprint): boolean
+ isNearDuplicate(Fingerprint, double): boolean
+ getDistance(Fingerprint): double
Extends
1
1
<<Interface>>
HashGenerator
+ generateHash(String): int
FeatureShingles
+ SizeType: enum
- type: SizeType
- size: int
+ FeatureShingles(int, SizeType)
- getFeatures(SizeType, String): String[]
- getChars(String): String[]
- getWords(String): String[]
- getSentences(String): String[]
1
1
<<Interface>>
NearDuplicateDetection
+ generateFingerprint(String): Fingerprint
+ getFeatures(): List<FeatureType>
+ getThreshold(): double
+ setThreshold(double): void
+ setFeatures(List<FeatureType>): void
Extends
XxHashGenerator
- xxhash: XXHash32
+ XxHashGenerator()
Extends
Extends
NearDuplicateDetectionBroder32
- features: List<FeatureType>
- threshold: double
- hashGenerator: HashGenerator
+ NearDuplicateDetectionBroder32(HashGenerator)
+ NearDuplicateDetectionBroder32(double , List<FeatureType> , HashGenerato
NearDuplicateDetectionCrawlHash32
- features: List<FeatureType>
- threshold: double
- hashGenerator: HashGenerator
+ NearDuplicateDetectionCrawlHash32(HashGenerator)
+ NearDuplicateDetectionCrawlHash32(double, List<FeatureType>, HashGenerator

Figure 3: Class diagram of the duplicatedetection package developed for Crawljax

The isNearDuplicate-method uses the getDistance-method and compares that value to the threshold and returns a Boolean-value. If the value of the getDistance-method is less or equal to the threshold, than the two fingerprints are duplicates. In *DuplicateDetection* the fingerprints can be generated from the content of a state with the generateFingerprint-method. This gives us the following steps to determine in two states are duplicates, which is also shown in figure 4. First get the DOM of the states. Secondly, strip the DOM to get only the relevant content of the state using the DOM-strippers available in Crawljax. Finally, generate the fingerprint using getFingerprint, which uses Features provided to the *DuplicateDetection*.

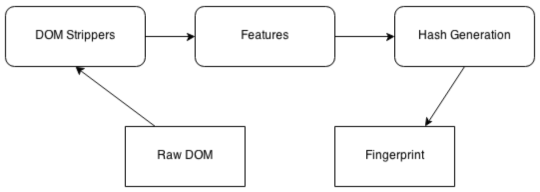


Figure 4: Process for generating a fingerprint

### 9.1.2 Hamming distance

In the Crawlhash-algorithm, the initial algorithm for the hamming distance of two fingerprints was just counting the number of ones in the exclusive-or operation of both fingerprints, see algorithm 3. The exclusive-or operation sets every bit-position where the bits of the two fingerprints are different to a one and sets a zero if both bits are the same. The resulting distance is equal to the number of ones in the hash generated by the exclusive-or.

**Algorithm 3** hammingDistance-function in Crawlhash

```
  Pre: Two fingerprints
  Post: Integer of the number of differences in the fingerprints

  int exOr ← fingerprintFirst ˆ fingerprintSecond {The exclusive or operation}
  String exOrString ← represent exOr as a bitstring
  int distance ← 0;
  for all bits in exOrString do
    if bit is a 1 then
      distance++
    end if
  end for
  return distance
```

After some research, a faster algorithm was found to calculate the hamming distance. This algorithm is an optimization of 3, which only uses bit-operations. This optimized algorithm was many factors faster than the original. See this blogpost[8] for a description of this algorithm. We extensively tested the algorithm though unit-tests to make sure that the optimizations of the algorithm did not result in broken functionality, but only in the performance. The results of the unit-tests showed that the optimized algorithm did have the same functionality as the original implementation.

### 9.1.3 Jacquard-coefficient

In the Broders-algorithm, the distance of two fingerprints can be found by calculating is the intersection of two sets and dividing it by the union of the two sets. The resulting value is called the Jacquard-coefficient. The two sets will be given in the parameters of this function. For calculating the union and intersection, we made use of the package com.google.common.collect.Sets from the Google Guava project[9].

## 9.2 Crawljax Calibration Tool

In this section the implementation of the calibration tool will be explained. First, the noteworthy implementation details are explained in section 9.2.1, including the used design

[8]http://yesteapea.wordpress.com/2013/03/03/counting-the-number-of-set-bits-in-an-integer/

[9]http://guava-libraries.googlecode.com/svn/tags/release04/javadoc/com/google/common/collect/Sets.html

patterns. Finally, the necessary external dependencies are identified in section 9.2.2.

### 9.2.1 Implementation details

The testing suite has some notable implementation details. First, the used design patterns are identified. Afterwards, the style conventions used in this system are regarded.

**Design patterns**
To help any future developers understand the source code quicker, the testing suite implements recognizable design patterns and common conventions.

Using Guice, a dependency injection library, all major classes make use of the dependency injection design pattern. The dependency injection design pattern is based on the Hollywood-principle: "Don't call us, we will call you.". Rather than having a class creating the needed objects at run-time, the required objects are passed into the class externally. This increases the testability and centralizing the binding of implementations to interfaces.

Another design pattern that is used frequently throughout the project is the Data Access Object (DAO) design pattern. This design pattern is responsible for decoupling the retrieval and manipulation of data from the actual storage method. A object using a DAO, does not care or even know what storage type is used underneath the DAO. In the calibration tool this design pattern is used with every kind of interaction with the database. Using this approach, the non-DAO code is not cluttered with any raw SQL-code and one can easily replace the database-DAOs with another storage-type.

Other design patterns that are used in a couple of situation, are the factory and singleton design patterns. These are used to respectively avoid implementing concrete classes and reduce unnecessary class-instantiations.

**Style Conventions**
To keep the code on a high stable level, the project uses the same code conventions as the Crawljax-core. Most of these conventions are similair to ones preferred by the Google Style Guide[10]. The project uses the common naming and style conventions, and uses interfaces where potential extensions of the program could take place.

In Java there are two types of exceptions. Checked exceptions should be caught or thrown on every level, while runtime exceptions, like the infamous NullPointerException, do not have to be caught or explicitly re-thrown. According to the book Effective Java[11], one should use checked expections for recoverable conditions and runtime exceptions for programming errors. As the custom exception we throw, such as the AnalysisException, are

[10]http://google-styleguide.googlecode.com/
[11]http://www.amazon.com/dp/0321356683/?tag=stackoverfl08-20

considered unrecoverable we implemented them as runtime exceptions.

### 9.2.2 External Dependencies

To adhere to the software principle of using tried and tested libraries where possible, we have several dependencies to external code. First of all, we use Maven for build automation and dependency management. Next to that we have the obvious dependency to the Crawljax-core module, to actually execute our crawls. ini4j is used to enable the reading of ini-style configuration-files, which offers more functionality compared to the native Properties-library in Java. The project uses the Mockito-library and the JUnit to enable easy (unit) testing of the project. The ORMLite library is used to enable Object-Relational Mapping (ORM) with the database, which removes lots of native SQL code from the java code. Hibernate was considered to heavyweight for this project. The project makes use of Lombok to decrease the amount of redundant code and increase the readability of the code. The Google Commons and Apache Commons libraries are used for some useful classes, such as immutable lists, immutable maps and argument interpretation. The project uses Logback under the SL4J Facade for logging purposes. Finally Guice is used throughout the project to make use of dependency injection, thereby increasing the testability and centralizing the binding of implementations to interfaces.

## 9.3 Threshold-slider

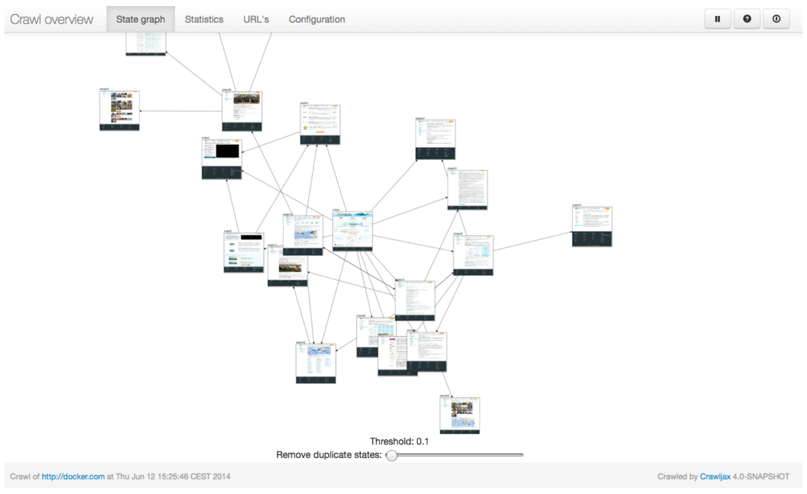
As explained in the threshold-part in section design, two states are labeled as duplicates depending to the threshold. To be consistent, this will be done by comparing the threshold with the getDistance method, which will return the distance between two states. Two states will be duplicates if the getDistance is lower than the threshold.

### 9.3.1 Dealing with getDistance

For CrawlHash, the distance can be an integer between the 0 and 32. The getDistance method will also return a value between the 0 and 32. This value represents the number of bits that are different in the fingerprints of both states. By using a threshold of 3, we actually say that as long as two states that have a difference of 3 bits the hash they are marked as duplicate states.
For Broder, the distance can be a value between the 0 and 1, because the Jacquard-Coefficient is be used to calculate the correspondence. See section 8.1.3 for the description of the Jacquard-coefficient. The Jaccard-coefficient gives the similarity-distance, where a high value will be returned if both states have much in common. To keep return the difference-distance between two states, i.e. a high value if both states are very different. So we need to invert the value. This can be easily done though one minus the result of the Jacquard-Coefficient.

### 9.3.2 Screenshots

Below are two screenshots of the crawloverview-plugin with the threshold-slider. Figure 5 is the default situation where the slider is situated at the leftmost value. This value is the initial threshold that was used for the crawl. Notice that lowering this value has no use. Lowering the threshold in the Crawloverview-plugin should probably result in showing states that are not crawled, because a lower threshold will make it less likely that two states will be seen as duplicates. Two states who were duplicates in the crawl may be not seen as duplicates with a lower threshold. The state was not added to the state-flow-graph during the crawl, so it is not possible to show the state in the crawloverview when the threshold is changed to a lower value. For this reason the threshold can only be changed to a higher value in the crawloverview. Figure 6 shows what happened when the value in the threshold-slide is moved to the right, thus increasing the threshold.



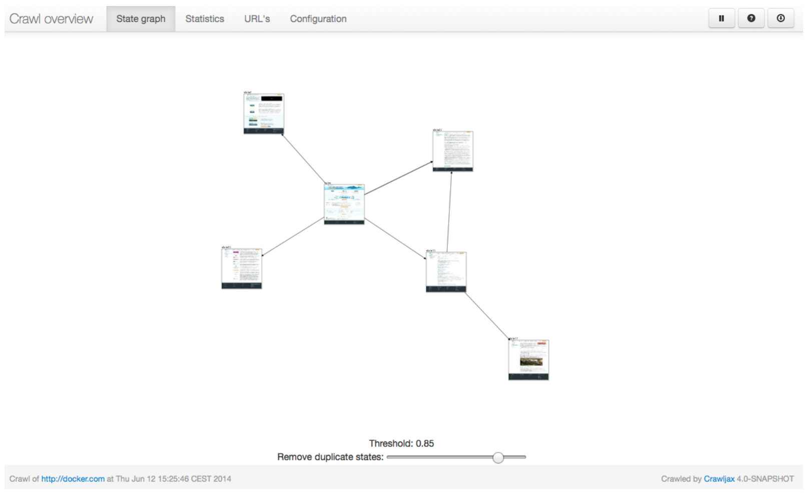
Threshold: 0.1

Remove duplicate states:

Crawl of http://docker.com at Thu Jun 12 15:25:46 CEST 2014

Crawled by Crawljax 4.0-SNAPSHOT

Figure 5: Threshold-slider at its default position



Threshold: 0.85

Crawl of http://docker.com at Thu Jun 12 15:25:46 CEST 2014

Crawled by Crawljax 4.0-SNAPSHOT

Figure 6: Threshold-slider moved to the right

### 9.3.3 Algorithm of the Threshold-slider

In the design of the Threshold-slider the method removeUnconnectedStates() was mentioned. During the design, not much attention was directed at this function, as it seemed trivial. However to make the algorithm more efficient the original algorithm has been re-factored to the following algorithm 4:

**Algorithm 4** Changed callback of the threshold-slider

```
statesInGraph ← empty array
for all states in states do
  for all distance in state.duplicateDistance do
    if state can be reached from the index then
      removeIfDuplicate(state)
    else
      remove state
    end if
  end for
end for
```

The two checks in the if-statement in algorithm 2 in the design-part will be performed in the method removeIfDuplicate. Note that this method should only be performed if the state is connected to the graph; there should be a path from the index to that state. If the state is not connected, the state can be deleted. In such a case it does not matter whether it is a duplicate state or not.

# 10 Code quality

A critical aspect of the project is the code quality. As mentioned in the requirements, a high code quality has been stressed by the client. The code should be of at least the same level of quality as Crawljax. In this section the different procedures that were used to ensure and improve the code quality.

## 10.1 Feedback SIG

As a part of the general structure of the final bachelor project, the code had to be send to the Software Improvement Group (SIG). At SIG, the software would be reviewed and ranked based on the software quality. The feedback on the code should give some hints on how to improve the software quality.

While we were still implementing new functionality, we sent the code so far to SIG on June the 13th.

Initially we were disappointed by the lack of specific feedback. We expected to receive all kinds of scores on various code quality aspects, so it would be clear on which aspects the code quality had to improve. Instead we received feedback stating that our code was "4 out of 5 stars", which indicates that the code is of "above average" quality (where 3 stars is the current industry state of practice). Additionally, the feedback included critique on the number of parameters required in a function in the calibration tool. However, this feedback seemed to be a misinterpretation of the code, as it was irrelevant in the context of the code.

It was however a clear indicator that the code in some places was not as obvious as we expected. Therefore, considerable effort was made to improve method-naming, class-naming and java-docs. Additionally, some inconsistencies in the class APIs were fixed.

## 10.2 Static Code Analysis

We used additional code quality analysis tools to obtain feedback on the code quality. CodePro Analytix[12] was used for this purpose. CodePro is a static code analysis tool maintained by Google. It uses highly extendable metrics to check on various code quality aspects, including code coverage, conventions, bad practices and dependencies. Another tool that was used was FindBugs[13], which is an open-source project maintained by the University of Maryland. Findbugs is quite comparable to CodePro, although it checks for some different bugs and flaws. Finally, EclEmma[14] was used to track the code coverage of our test-cases.

These tools provided a lot of useful feedback, about minor inconsistensies in the code. These minor inconsistencies include bad naming practices, magical numbers and attributes missing the *final* indicator. We fixed nearly all kinds of inconsistencies we deemed relevant.

[12]https://developers.google.com/java-dev-tools/codepro/doc/
[13]http://findbugs.sourceforge.net/
[14]http://www.eclemma.org/

Therefore, the code adheres to all popular Java code and style conventions. Next to this feedback, CodePro allowed us to view the dependencies of a package in a visual way, shown below:

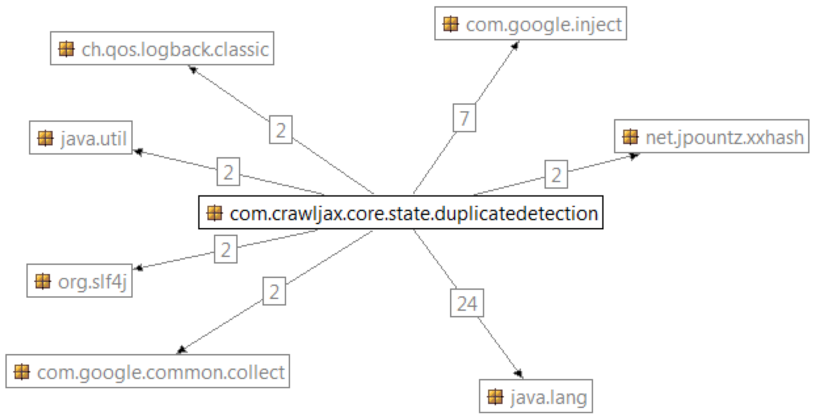


Figure 7: Dependency graph of the duplicatedetection-package in Crawljax

As can be seen in the graph of figure 7, the duplicate-detection package in Crawljax does not depend on any package inside the Crawljax-project. By adhering to this *good practice* of limiting the inter-package dependencies, it is very simple to reuse this package in any other project without having to worry about breaking anything inside the package.

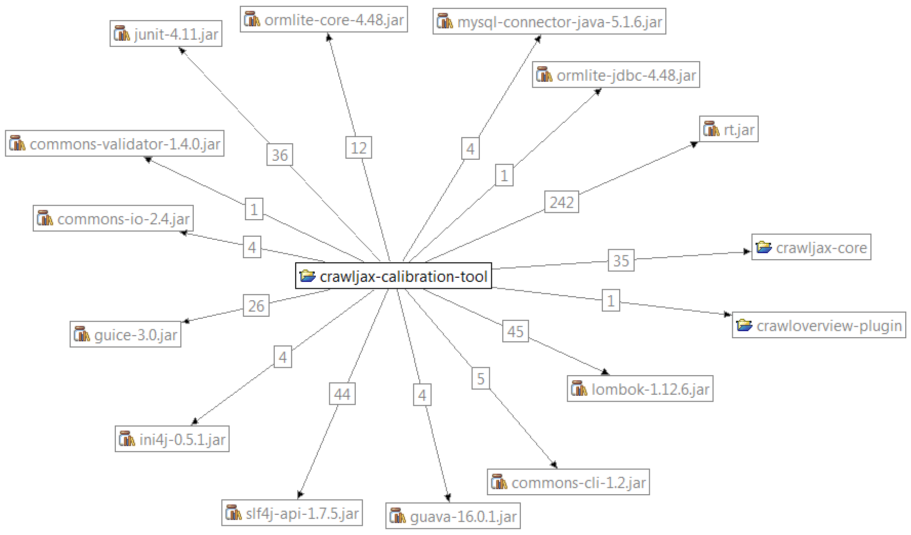


Figure 8: Dependency graph of Crawljax Calibration Tool

In the calibration tool-project dependencies on Crawljax are essential. The dependencies are however limited to just the modules that are actually needed, namely Crawljax-core and the Crawloverview-plugin. The other dependencies have all been assessed, whether they are actually needed. For all remaining dependencies in the graph this is the case. Figure 8 shows the resulting graph of the dependencies in the calibration tool.

### 10.3 Code coverage

Inherent to a project concentrated on improving a testing tool, is the need for tested code. As mentioned in the methodology, a test-driven development (TDD) approach was used in this project. Therefore, the code has a extensive set of tests. The current code coverage is 97.4% for the code added to Crawljax and 75.5% for the testing suite.

Using the tools and feedback throughout the project, the code quality has been kept up to a high standard. The code follows common Java code and style conventions, is thoroughly tested and follows the conventions used by the original Crawljax Project.

## 11 Results from the calibration tool

At the moment, we have introduced two duplicate detection algorithms: Crawlhash 8.1.2 and Broder 8.1.3. Both algorithms need a certain feature-size, feature-size-type and threshold. In this section we will test both algorithms on as many different values as possible for these parameters. The results of Crawlhash on different parameters will be shown in section 11.2 and the results of Broder in section 11.3.

## 11.1 Test-setup

To test the algorithms, we will run Crawljax on a training-set of 6 websites. These websites will be annotated to get the optimal result for the duplicate detection. How this is done is shown in section 8.2.4. Next we will run our analysis tool on these 6 websites for different parameters. The analysis will get the difference between the crawl and the optimal result, see section 8.2.3. The 6 website have a total of 119 states, where 51 are unique states. We tested both algorithms on this training-set for the three shingle-size-types: chars, words and sentences.

The old duplicate-detection in Crawljax gave an average of 11 false negatives and 0 false positives, so a total of 11 error. Because this algorithm made no use of parameters, it will give the same result, 11, for all the different parameters. For reference, the plane with value 11 will be shown in green in every graph of a new near-duplicate detection algorithms. To get a better visual view, some figure are shown for two different viewpoints.

We tested the threshold to the value where the number of missed stated became higher than the total error on several other values. Two stated have a higher probability to be duplicates when the threshold increases. So the missed states will grow if the threshold increases and make the results only worse. For the feature-size, we try to test as many different values until we see some trend in the results.

## 11.2 Results of Crawlhash

This section shows the results of the performance of Crawlhash on Crawljax. The number of mistakes are plotted into figures. A mistake is defined as: Crawlhash marks two states as duplicates, but the states are actually not duplicates (false positive) or Crawlhash does not mark them as duplicates, but they are actually duplicates (false negatives).

**Shingle-type: words**
Figure 9 shows the result of the total error that Crawlhash made on different values for the feature-size and threshold for the feature-size-type 'words'.

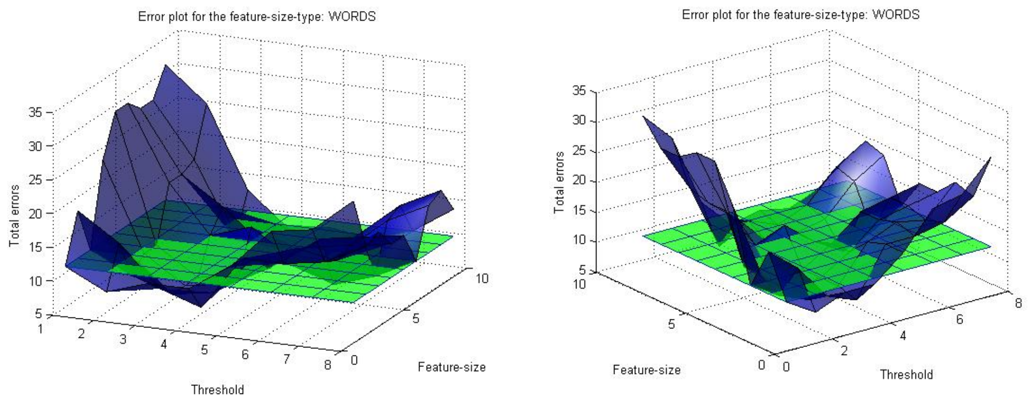


Figure 9: Result of Crawlhash for feature-size: 1-9 words and threshold 1-8 for two different viewpoints

One can see that there is a big area for very low and very high thresholds that is above the plane. These thresholds will generate more mistakes than the current version of Crawljax, so we should concentrate on the values below this plane. For the lowest feature-size, the threshold of 2 to 4 this will give nice results. For higher feature-sizes, this threshold will shift a bit to the right. This gives a threshold between 4 and 6 as good results for the feature-size 9.

**Shingle-type: sentences**
The result for the feature-size-type sentences is shown in figure 10

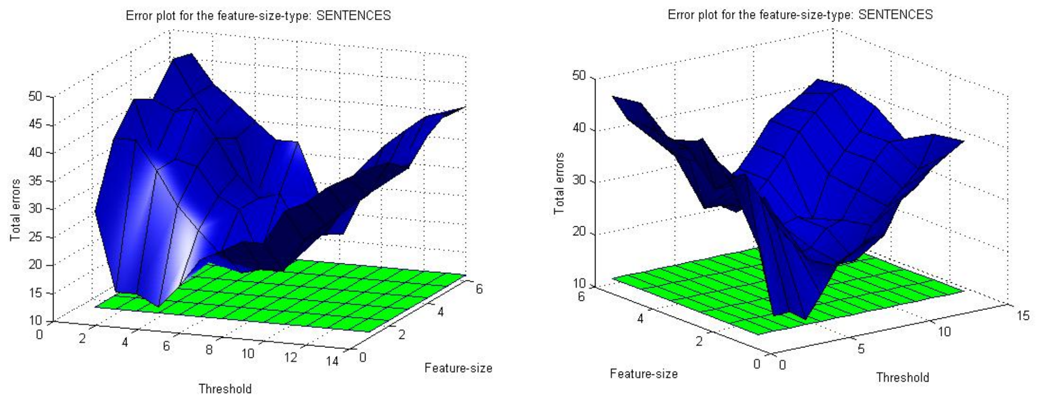


Figure 10: Result of Crawlhash for feature-size: 1-6 sentences and threshold 1-14 for two different viewpoints

The figure shows the same curve as the results for the words, but in this case, all the values are above the plane. This means that Crawlhash has made more error than the current duplicate-detection algorithm in Crawlhash for every tested combination of parameters.

**Shingle-type: chars**
The result for the feature-size-type sentences is shown in figure 11.

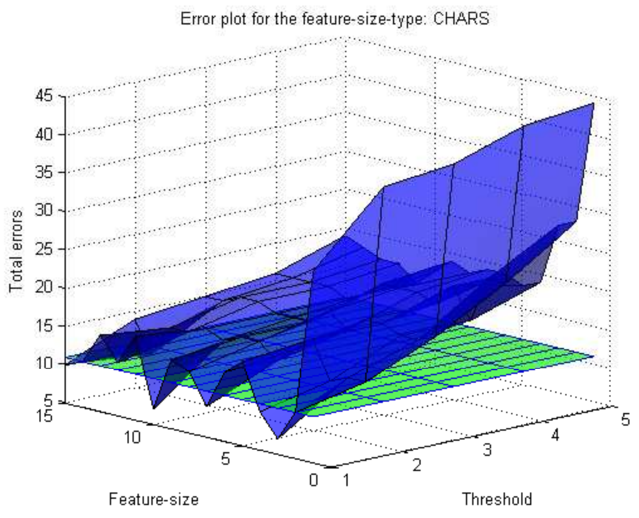


Figure 11: Result of Crawlhash for feature-size: 1-15 chars and threshold 1-5

**Conclusion Crawlhash**
The 'sentences' are not useful, because they had many more errors than the current duplicate-detection in Crawljax. The 'chars' are unreliable, because a small change in the threshold can lead to many unexpected changes in the number of errors. But the 'words' are useful for certain thresholds. This optimal threshold shifts to higher values for higher feature-sizes. Figure 12 shows all graphs in a single figure.

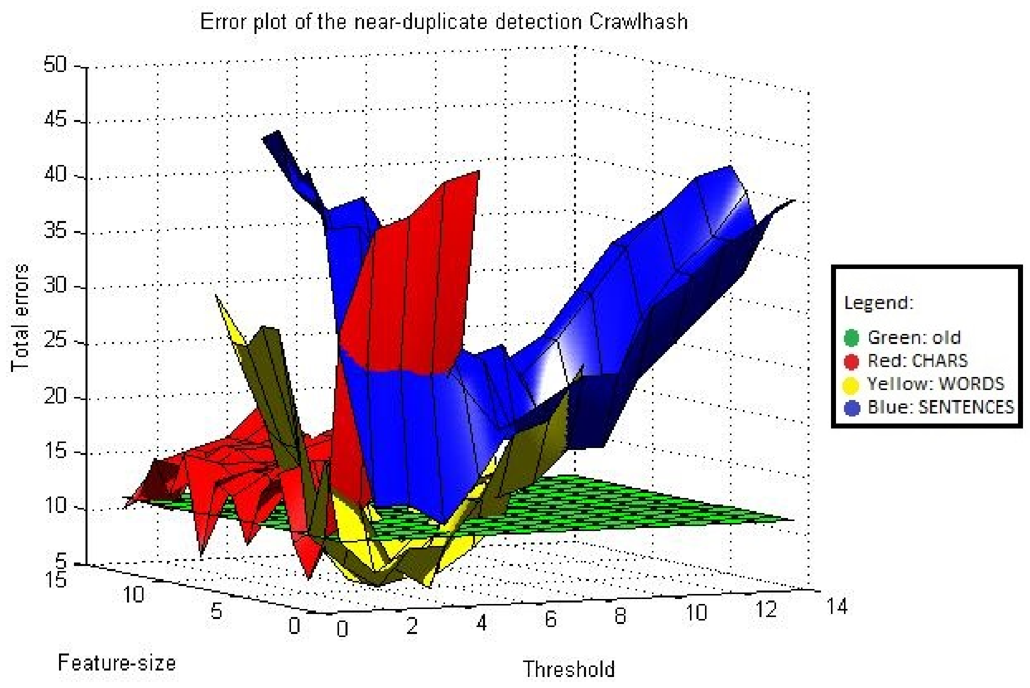
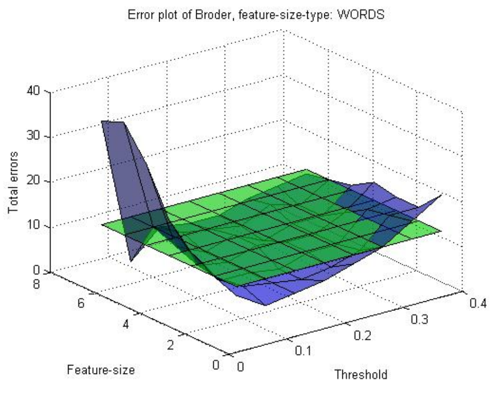


Figure 12:

## 11.3 Results of Broder

The results of Broder near-duplicate detection algorithm will be shown in this section in the same way as the results of Crawlhash in section 11.2. Any value between 0 and 1 can be chosen for the threshold for the fingerprints of Broders algorithm . We decided to use step size of 0.05 to get a good overview of the performance of the algorithm, but also limit the number of tests to make it possible to run the tests in a reasonable amount of time. Additionally, the maximum threshold and maximum size of the shingles was limited to reasonable values. When results were consistently getting worse, which often happends on large thresholds and large shingle sizes, testing was stopped.

**Feature-type: words**
Figure 13 shows the results for the 'words' type of shingles on two different viewpoints.



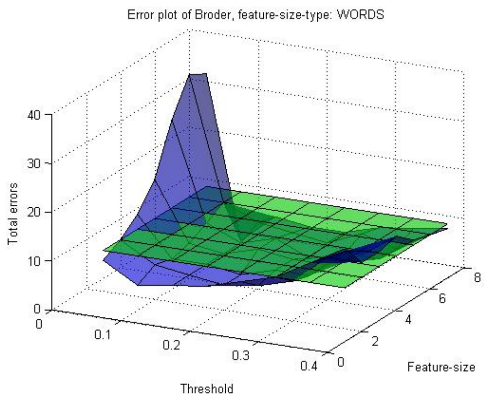


Figure 13: Result of Broder for feature-size: 1-7 words and threshold 0.05-0.4 with steps of 0.05 for two different viewpoints

First thing to notice is that the figure is way more smooth than the figures of Crawlhash, even though the step size is bigger for Broder (Broder: 0.05 and Crawlhash: 1/32≈0.03). See the disadvantages of Crawlhash in section 8.1.2 for possible explanations of this behavior. Secondly, there clearly is a large area of threshold and shingle-size combinations that substancially outperforms the original duplicate-detection of Crawljax.

**Feature-type: sentences**
Figure 14 shows the results for the feature-size-type 'sentences'.

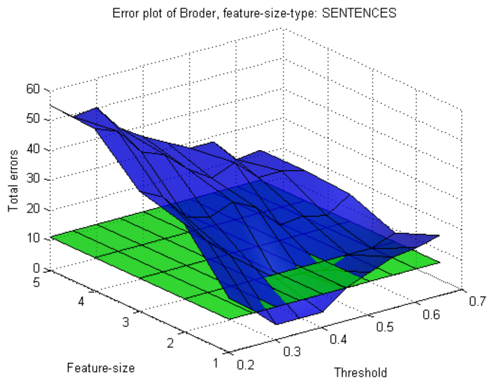


Figure 14: Result of Broder for feature-size: 1-5 words and threshold 0.2-0.3 with steps of 0.05 for two different viewpoints

What can be noticed from the graph above is that only the results of the feature-size '1-sentence' have some threshold-values that outperform the original duplicate-detection algorithm. All other values provide worse results than the current version of duplicate-detection in Crawljax. A possible explanation for these results is that the average web-page only has too few sentences to really generate a representative hash.

**Feature-type: chars**
Figure 15 shows the results for the 'chars' shingle-type.

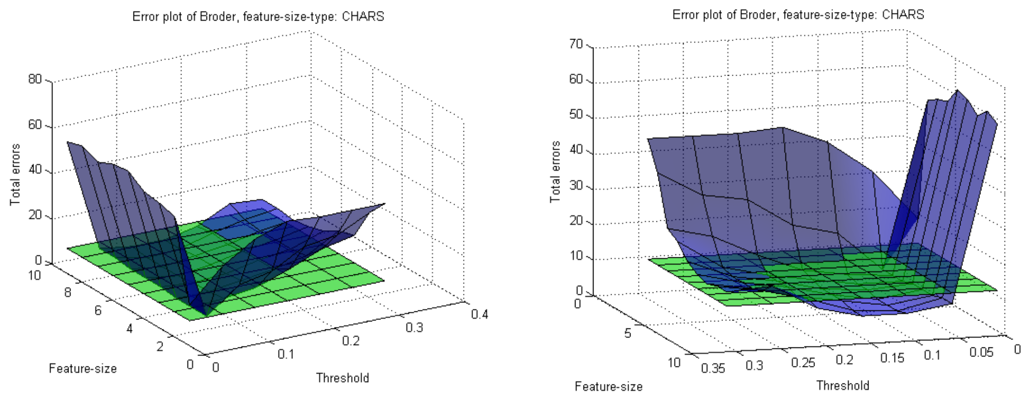


Figure 15: Result of Broder for feature-size: 1-9 chars and threshold 0-0.65 with steps of 0.05

A similar pattern like the sentences shingle-type can be seen in this figure; bad results for very high and very low threshold values. Important to note is that a feature-size higher than 3 chars will give nice results for a threshold value between 0.05 and 0.15.

**Conclusion**
In comparison with the Crawlhash results, the result of Broder looks more reliable. The graphs show that there are large areas of threshold and feature-size combinations for words and characters feature-types, that outperform the original duplicate-detection implementation. Especially, the 'words' feature-type has a lot of combinations that substancially outperform the original implementation.

## 11.4 Conclusion of the test-results

Most of the test-results shows a curve of a parabola that opens upward, which shows us that the algorithms have some optimal values for the parameters that are best. The best results in this case are the results with the least number of errors, where a missed state count as one error even as a duplicate state. A very low threshold results in way to many states, so the result has a lot of duplicate states. A very high threshold results in to less states, to many states were seen as duplicates. This explains the worse results for low and high thresholds. To get nice results, above figures shows that a value in between can give very nice results if we look at the total number of mistakes. If duplicate states are prefered above missed states, some smaller value than the optimal can be used and if missed states are prefered, some higher value can be used. This shows that if will be very helpfull to know the value of the threshold which gives the least amount of total error made by the algorithm.

Another thing we noticed is that the optimal value for the threshold shifts to a slightly higher value when the feature-size increases, but the number of mistakes are almost the same.

The task in now to find out if the optimal value for the threshold will outperform the other values for the parameters on most different web applications, or that this value is dependent on certain aspects, such as the DOM-size.

## 12 Discussion and Recommendation

There are some decisions we had to make, which we want to discuss in this section. The first decision is what parameter values should be used and how we should define these parameters. One question can be: May the user change these parameters? The decisions according to the parameters will be discussed in section 12.1.

### 12.1 Parameter values

For the near-duplicate detection are two parameters necessary. The first one is for the features. This contains the feature-size-type: chars, words or sentences and a feature-size, i.e. the number of chars/words/sentences. The second parameter is for the threshold to decide when two states are duplicates of each other. We have considered the following options to handle these parameters:

- Find a robust or optimal values and always use these values for the parameters
- Let the user decide the parameter values before each crawl
- Use default values that gives understandable results and make it possible to change these values by the user

While testing the near-duplicate detection algorithm, we noticed that it was hard to find one single optimal or robust value for the parameters. This makes the change to high to get bad results for a crawl for some websites. At the moment we do not have such a optimal value, so we cannot decide for the first option. The people who use Crawljax are most of all people with technical knowledge, so letting the user decide the values for the parameters is not a strange idea. But the problem is the first use of Crawljax. People who use Crawljax for the first time will just run it on some web-application for there purposes, for example, testing the web-application or finding all states to make it reachable for search-engines. They probably do not want to bother about some parameter values for near-duplicate detection the first time they use Crawljax. So leaving the decision to the user, as we proposed in the second option will not be a good solution. This brings us to the third option, where we predefine the values for the parameters, but make it possible for the user to change these values. A user of Crawljax can first use Crawljax without bothering about the parameter values, but if the result of the crawl is not satisfactorily, he will be able to find the problem and change the parameter values according to his needs.

Now we decided on the third option, we need to define the values of the defaults. With our trainings-set we have tested all relevant combinations of values for the parameters. By selecting some values which gives nice results and test these on other, randomly selected, websites, we get the default values for the parameters. By selecting some values with nice results, we prefer duplicate states above missed states, because it is clear what to do if there are to many duplicates in the crawloverview. The threshold-slider can also be used to see what would have happened when the user had run Crawljax with a higher threshold. But if a crawl will give way to many states, a first-time user will probably not know what went wrong.

We recommend to use the default values for the parameters and change them if the result of the crawl is not satisfactory. Lower the threshold if there are to many states missing and make the threshold higher if there are to many duplicate states.

## 13 Conclusion

Crawljax wants to find all relevant states of a web application. For a large web application it can take a lot time to find all these states. To limit this number of states that Crawljax should visit, we want to get rid of duplicate and near-duplicate states. Identifying near-duplicate states at run-time makes it possible to merge the newly found state with an existing state. The advantage of this is that near-duplicate states do not have to be crawled and analysed, because it is a near-duplicate of a state which is already been crawled and analysed. This makes it possible to crawl a web application in the same amount of time.

The problem of identifying near-duplicate states is an optimization problem, for which no optimal algorithm exists. As described in the problem analysis, this is due to inability to consistently identify unique content from near-duplicate content.

The main goal of this project was to build a near-duplicate detection algorithm for Crawljax, which should consistently perform better duplicate-detection than the original duplicate-detection of Crawljax. The product should be seamlessly integrated into Crawljax offering the user various options to configure the duplicate-detection. On the other hand the product should change as little as possible to the existing API of Crawljax.

The final product of this project satisfies all requirements. As shown in the test-results. There are several configurations of the duplicate-detection which outperform the old duplicate-detection consistently. To serve users with more options, two algorithms are available, namely Crawlhash and Broder. Crawlhash is faster and costs less memory than Broder. Broder on the other hand is more consistent and provides better results.

The product is highly customizable by users. Besides the choice between the two algorithms, users can customize nearly everything about the algorithms. How the hashes are generated. What features from the web-page should be taken in account when generating the fingerprints. How strict system should be when deciding whether a state is a duplicate, using a threshold-value.

Finally, to visualize some of the magic going on in the duplicate-detection-package, the so-called threshold-slider is included in the final product. As explained in the implementation-section, with this slider the user can view the state-flow graph under different threshold-values.

This functionality does not target inexperienced users who are unfamiliar with the concept of near-duplicate detection. For those users sensible defaults have been implemented. However, for the experienced users this functionality offers better duplicate-detection and more customization possibilities, which in the end increases the added value of Crawljax.

Based on the results from the calibration tool, we could say that it is possible to limit the number of states that needs to be crawled to get an exhaustive crawl by using a near-duplicate detection algorithm. When correct parameter will be used, it is possible to find all states on a web application, while regarding all duplicate states.

# 14 Future work

Although the deliverables of this project provide a strong basis for near-duplicate detection algorithms, there are still a lot of features that could potentially enhance the duplicate-detection.

## 14.1 Feature based on outgoing links

Different feature types can be used in the future. Looking at the outgoing links to decide if a state is a duplicate state can be a nice idea, because in that case you are sure that the outgoing links of that states are already visited if the state is marked as duplicate. With the textual near-duplicate detection it can be possible that two states have the same content but the first state has less outgoing links than the second state. This results is not crawling the links that are in the second state but not in the first state, because the second state will be marked as duplicate so its candidate clickables will not be crawled.

## 14.2 Genetic Algorithm for the calibration tool

Using a genetic algorithm to find the optimal default values for the parameters of the near-duplicate detection algorithm can result in better results, easier testing for optimal values and less time spend for finding optimal values.

# References

[1] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[2] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[3] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. 1999.

[4] Monika Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 284–291. ACM, 2006.

[5] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150. ACM, 2007.

[6] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 122–134. IEEE, 2008.

[7] Ali Mesbah and Arie Van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, pages 210–220. IEEE Computer Society, 2009.

[8] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3, 2012.

# Appendix A: Survey of duplicate-detection algorithms for Crawljax

Erwin van Eyk Wilco van Leeuwen

Delft University of Technology
{E.D.C.vanEyk, W.J.vanLeeuwen}@student.tudelft.nl

April 2014

**Abstract**

On the web near-duplicate documents are abundant. As many as 40% of the pages on the Web are near-duplicates of other pages, according to Manning et al. [10]. A web crawler should be able to recognize and deal with near-duplicate web pages.
In this survey we will first explore the most prominent duplicate-detection algorithms, which could be viable implementations in Crawljax. The most prominent algorithms which could provide a solution are Charikar's Simhash algorithm [4] and Broder's shingling algorithm [3]. Secondly, we will identify the requirements for implementing the duplication-detection algorithms. Additionally, requirements will be established for a testing platform to be able to objectively benchmark the various duplication-detection algorithm implementations.

**Keywords**: fingerprinting, Crawljax, near-duplicate, document similarity, hashing, features

## 1 Introduction

Web crawlers attempt to find all the states of a provided domain, which can be a single website up to the entire Internet. There are several motives to run a web crawler. Google indexes the web to make it easy for people to find relevant websites on the entire web [2]. The Internet Archive[1] attempts to save all information on the web to make it accessible to everyone, even when the content on the original site has be removed or lost. Because of the extremely large size of the entire web, currently estimated to contain 3.1 billion static web

[1]http://crawler.archive.org/index.html

pages[2], efficient algorithms are needed for web crawlers. A lot of these web pages or documents can be duplicates of near-duplicates of each other. These duplicate and near-duplicate web pages create problems for web search engines: They increase the space needed to store the index, slowing down the crawlers and increasing the cost of serving results, and annoy the users [7]. It is possible that the advertisement on a state changes, with the result that the state will be seen as a new state while all the rest is the same. Another example is a counter on a page that keeps track of the number of visits to that state. The counter is different every time the crawler visit that state, so the crawler will continuously see the state as a new state. Eventually the crawler will not finish crawling, because it can finds an infinite number of 'new' states.

Crawljax[3] is a specific type of web crawler. Originally developed by Mesbah et al. [11], it is a unique open-source web crawler with the purpose of automatically testing websites for errors, inconsistencies and broken links. An important feature of this crawler is the fact that it can deal with Javascript to search for hidden links in the so-called *hidden web* [16]. Crawljax initially shipped with a very basic duplicate-detection algorithm [11]. The algorithm compares the complete DOM of one state to other states using Levenshtein method. This basically comprises a string comparison. The number of different characters is compared to a threshold. If the difference between the two states is smaller than the threshold, the two states are considered to be equal. This algorithm fails to recognize a chunk of the near-duplicate states. Additionally, this method is quite expensive in terms of computation, because it compares each state with each other state for each character of the whole DOM. The DOM can be very great for large and complex states.

Currently, Crawljax makes use of string comparison on a normalized DOM. The normalized DOM contains only the HTML-attributes of the DOM. This is already much less expensive in terms of computation, but still is unable to detect most near-duplicate states.

Implementing a more efficient way of detecting near-duplicates can result in a better performance. In section 2 we will first discuss what a crawler is, than explain the difference between Crawljax and other crawlers in section 3. In section 5 the problem of duplicates will explained for web crawlers and in section 6 some techniques are proposed tackle this problem with. Finally in the last sections we will list the requirements and details of the implementation of both the algorithm and the related testing environment.

## 2 What is a crawler

In order to understand the implications of the different fingerprinting algorithms in Crawljax, it is useful to have a clear definition and understanding

[2]http://www.worldwidewebsize.com/
[3]http://www.crawljax.com

of the functionality of a regular web crawler. A web crawler, also known as a (web) spider, searches the web for hypertext links, typically for the purpose of web indexing [2]. There are numerous implementations of web crawlers. Some of the better known crawlers are Googlebot[4], Mercator [8], Bingbot[5] and UbiCrawler [1].

## 2.1 Purposes of crawlers

The most well known use of web crawlers has always been indexing. Brin and Page [2] had ideas to use the index to make an effective search engine to make it easy for people to find useful document on the web. The first idea was to give every page a certain rank, whereby a higher rank would represent a more useful result. Another idea was to use anchor text in the index to refer to the page the link points to, in stead of the page where the link is on. This is useful, because (1) anchors often provide more accurate descriptions of web pages than the pages themselves and (2) anchors may exist for documents which cannot be indexed by a text-based search engine. With these ideas, Brin and Page started the search engine, Google, which eventually became the most popular search engine in the world.

Another common use of crawlers is to use them to test a website. It allows developers to automatically test their website as a black box. Crawlers build for this purpose, like Crawljax, will exhaustively crawl a website and documenting any errors, broken-links and inconsistencies found on the web pages.

## 2.2 Basic structure and components of a crawler

A crawler needs a list of seed URLs as its input in order to start crawling. It will remove an URL from the list, called the URL frontier, and determine the IP address of the host's name by a domain name system (DNS) look-up. Once the IP address is known, a connection with the server can be made and the corresponding document is downloaded. The crawler will search the document for links and add each link to the URL frontier if it is not encountered before. This is shown in algorithm 1.

The functions in this algorithm are the basic components of a crawler. URLs can be represented as a FIFO queue, where URLs.removeElement() always removes and returns the first element in the list and URLs.addElement(URL) adds the URL add the end of the URLs list. The common name of the list of URLs is *URL frontier*. Another list, called *visited*, keeps track of all URLs that are visited to avoid infinite loops. The notContains procedure will check if the list, *visited*, does not contain a certain link, so the link is not seen before. This component is often called *URL-seen test*.

[4]https://support.google.com/webmasters/answer/182072
[5]http://www.bing.com/webmaster/help/fetch-as-bingbot-fe18fa0d

**Algorithm 1** Basic crawler

```
Input:  List l of URLs
Input ← l
URLs ← l
visited ← $\emptyset$
while URLs $\neq \emptyset$ do
  URL ← URLs.removeElement()
  adress ← getIPadress(URL)
  content ← download(adress)
  for all links in content do
    if visited.notContains(link) then
      URL ← getAbsoluteURL(link)
      URLs.addElement(URL)
    end if
  end for
end while
```

In this way, a crawler will only find pages that are reachable by hypertext links. In the next section we will look at Crawljax which also finds new pages/states that are only reachable by inspecting Javascript functions.

## 3 Difference between Crawljax and other crawlers

The most well-known search engines only index the pages that are reachable by hypertext links with a unique URL and ignoring search forms, pages that require authorization or prior registration [16] and client-side scripting [12]. This is called the *publicly indexable web.* As mentioned in the introduction, the web content behind forms and client-side scripting is referred to the *hidden Web.* Crawljax differ from the well-known search engines, because it can detect dynamic contents of AJAX-based Web applications automatically without requiring specific URLs for each Web state [12].

Nederlof et al. [13] used Crawljax also for a testing purpose by applying the W3C validator to the structure of every DOM/HTML state. In this way, all states, even states in the *hidden web* can be tested for errors and warnings in the HTML code.

## 4 Problem Analysis

Crawlers have to crawl a lot of states to complete crawls of big websites. Such a website can contain a massive amount of different states. So it is important for a crawler to detect duplicate states if the state is crawled before, that it not crawl unnecessary states. Duplicate states occur because (1) many states have links that refer to the same content and (2) many different states have the same content. When a crawler does not take care of these problems, it will end up crawling a lot of unnecessary web states and showing a lot of unnecessary states in the result of a crawl. To avoid crawling an already visited state, a list will be retained with visited URLs. This can solve the first problem where many pages refer to the same URL, if the URL are different for each different state. This is true for the static web. But with AJAX it is possible that two completely different states have the same URL. The second problem, where the same content is on different URLs, emphasizes the previous problem. Namely that even without the use of AJAX, duplicates can occur, that cannot be solved by simply crawling all different URLs. Different URLs can also contain duplicate states. To detect these duplicates, the document should be downloaded before the content can be used for a test to check on duplicates. The state will be seen as a duplicate if the content is already been found earlier. The comparison of the content of two pages is hard, because the content is mostly not exactly equal. Two documents can be identical in terms of content but differ in a small portion of the document such as advertisements, counters and timestamps [9].

## 4.1 Many pages refer to the same content/page

Crawlers must detect whether a link has already been visited. Mercator [8] uses a URL set containing all the URLs seen. Before some URL is added a URL-seen test is performed. To make this scalable, it should be time and space efficient. To save space, the URL is saved as a fixed-sized checksum and to save time, Mercator keeps an in-memory cache of popular URLs and recently added URLs. But several different URLs can point to the same location, so it can be efficient to compare the IP addresses instead of the URLs. This will cause a bottleneck for the DNS lookup, because every URL must be mapped to its IP address. Heydon and Najorkmade [8] made a multi-threaded DNS resolver that can made multiple requests in parallel which solves this bottleneck. All this looks promesing for the static web, where every different URL or IP points to a new state. But with the technology called AJAX, this is not the case anymore. States found with an URL that is already present in the URL set should also be checked to see if the content different of the previous found states.

## 4.2 Same content on different pages

A lot of pages on the web contain the same information. There are many causes for the existence of near-duplicate data. For example typographical errors, versioned, mirrored, or plagiarized documents, multiple representations of the same physical object, spam emails generated from the same template, etc [17]. Detecting this similar content will result in ignoring the page, which saves network bandwidth, reduces storage costs and improves the quality of search indexes [9].

Mercator [8] uses a content-seen test to decide if a document had already been processed. It would be to expensive to save the whole content of every downloaded document, so it uses a document fingerprint set, that stores a 64-bit checksum of the contents of each downloaded document. Munku et al. [9] proposed to use simhash [4] to represent the page as a bit-string fingerprint and use hamming distances of the fingerprints to measure the similarity between pages. Similar documents will be hashed to similar hases, which makes it possible to compare the content of the document with the content of other document on the web.

## 5 Problem definition

In this paper we will focus on the problem of finding near-duplicate web pages. As mentioned in the introduction, the current state of detecting duplicate states by Crawljax is suboptimal. Due to the lack of effective duplicate-detection, this could lead to the state explosion problem[6]. There are several algorithms, which could improve the duplicate-detection, which we will describe in the next section.

## 6 Duplicate detection algorithms

To improve the performance of Crawljax, it is necessary to know that the contents of a web page have already been crawled. If a page is a near-duplicate of another crawled web page, it is likely that the candidate clickables on that page have already been crawled. If Crawljax detects a near-duplicate, it can ignore the web-page and the related clickables. Another advantage of near-duplicate detection is that two near-duplicate states will not both appear in the resulting state diagram of Crawljax. So near-duplicate detection can make Crawljax performance better, as long as the algorithm is efficient.

Duplicate detection can start from the moment that a crawler receives the HTML-code from a web page. All the unnecessary content of the HTML-code will be deleted, for example all HTML-tags and formatting instructions. This will results is a very large string of alphanumeric characters. In order to handle this large data, it will be split up in features. This separate can be done in several ways. We will explain some of these in section 6.1. Charikar [4] and Broder [7] found efficient ways to represent these features as relatively small bit string that can be used to find near-duplicates, which we will explain in section 6.2 and section 6.3.

### 6.1 Feature-selection

An important aspect of all near-duplicate-detection algorithms is the selection and weight-assignment of features. Features are low dimensional mappings of high dimensional properties of the document/state. In the rest of this section the general types of features are discussed.

[6]http://en.wikipedia.org/wiki/Model_checking

### Shingles

Nearly every established duplication-detection algorithm, including Broder et al. [3] and Charikar [4], explain their algorithms using words. They use a set of unique $w$-shingles to split the text of a document into features. A shingle is a subsequence, which can contain for example a fixed number of characters, words or sentences.

### Traditional IR techniques

On contrast with the use of shingles there is also the generation of features using regular Database Management System (IR) techniques. The idea is to compute "document-vector" of the document by case-folding, stop-word removal, stemming, computing term-frequencies and finally weighing each term by its inverse document frequency (IDF) [9].

### Connectivity information

Besides actual content of the page, the paper by Dean et al. [5] suggest using connectivity information as additional features. These features would require the crawler to basically compute a feature from all links from the page. This method is based on the idea that (near-)duplicate pages probably also have the same outgoing links.

### Anchor text and anchor window

Similar to the features based on connectivity information, Haveliwala et al. [6] suggests using anchor text and anchor windows as features. In this context an anchor-window is the neighbouring content of a hypertext-link. The idea behind this approach is that similar pages have the same descriptions for links.

### Document meta-data

Finally there is the meta-data of a web page. Attributes, such as the language-type, encoding and description, are often also similar for duplicate pages as Manku et al. [9] suggest. Combined with features mentioned in this section, these attributes could improve the duplicate-detection.

### Other forms of meta-data

Besides the regular meta-data from the document, it also possible to use the meta-data about the web page from other sources. Attributes about the web page taken from other or derived sources could be used as features. These could include, Search Engine Ranking or screenshots of the web page.

## 6.2 Charikar's Simhash algorithm

Currently Charikar's Simhash algorithm is one of the most popular near-duplicate detection algorithms [7]. The simhash algorithm [4] is a dimensionality reduction technique. It maps high-dimensional vectors to small-sized fingerprints [9]. Afterwards these fingerprints can be compared using the Hamming distance. A small Hamming distance indicates more similarity between two documents. In algorithm 2 we show the basic steps for the implementation of Simhash.

**Algorithm 2** Charikar's Simhash algorithm

```
define a constant $b$
define a vector V of size $b$
retrieve a set of features (e.g. word shingles) from the document
hash each feature using a b-bit hashing function
for each hash h do
  for bit i in h do
    if bit[i] of h is set then
      add 1 to V[i]
    else
      subtract 1 from V[i]
    end if
  end for
end for
for each bit in V do
  if $V[i] > 0$ then
    $V[i] \leftarrow 1$
  else
    $V[i] \leftarrow 0$
  end if
end for
```

An advantage of this algorithm is that the results, very small hashes, are easy and cheaply to compare to each other. Thus, for large datasets, which are often the case with crawlers, this method would result in less space reserved for fingerprints and faster similarity-comparisons between pages, according to Manku et al. [9] Besides the well-known Charikar's Simhash algorithm, there are also some closely related algorithms. These include I-Match and SpotSigs. On the other hand, the excellent comparison by Henzinger [7] showed that the effectability of simhash is suboptimal. Although simhash outperforms Broder's shingling algorithm overall, the shingling algorithm has a higher fraction of the duplicates found on the same site than simhash. As Crawljax mainly focusses on one website at a time, this would require some tweaking of the simhash algorithm to be as effective as the shingling algorithm.

### 6.3 Broder's shingling algorithm

As mentioned in section 6.2, Broder's algorithm has a higher fraction of the duplicates found on the same site than simhash. Henzinger [7] evaluated Broder's algorithm on 1.6B distinct web pages. She showed that 92% of near-duplicate pairs found by the algorithm of Broder belong to the same site. Simhash found only 74% of its near-duplicates on the same site. This makes the algorithm of Broder very attractive for Crawljax, because Crawljax is only interested in duplicates on the same site.

The algorithm of Broder uses $w$-shingles to divide the document into a set of continuous sequences of $w$ words [3]. All the sequences in the set are hashed with Rabin's fingerprinting method. This set of hashed features is used to compare the resemblance of two documents by calculating the ratio of the intersection divided by the union of the elements of the sets. This ratio is called the Jaccard coëfficient: $\frac{|S(d) \cap S(c)|}{|S(d) \cup S(c)|}$, where $d$ and $c$ are documents, $S(d)$ and $S(c)$ are the the sets of hashed features and $|S(d)|$ is the number of elements in the set $S(d)$. By using a threshold $t \in (0, 1)$, we can say that two pages are near-duplicates of each other when the ratio is higher that $t$.

To make this algorithm faster, Border et al. [3] described *super shingles*, which are computes by sorting the shingles and than shingling the shingels. When two document have one super shingle in common, so the documents have a sequence of shingles in common, then it is likely that the two documents are near-duplicates. Near-duplicate detection consist now only of finding a single super shingle that the documents have in common. This extra efficiency come to a cost. Short documents, which already have a few shingles, will only perform worse with super shingles, because a super shingle represent a sequence of shingles. The probability of a common shingles will be low in this case, which can lead to a lot of false negatives for short documents.

### 6.4 Locality Sensitive Hashing using bins

Another (although proprietary) method by Pugh and Henzinger [15] for finding near-duplicate documents, uses a method already widely used in the generation of checksums. The idea is that the words are splitted, and perhaps other features, into tokens. Afterwards the tokens are divided over k buckets (similair to md5). For each bucket a checksum is computed. The set of checksums or hashes of two apparently similar documents should agree for most of the buckets [9]. (e.g. by defining a threshold)

## 7 Testing environment

In order to have an objective comparison of the performance of the possible algorithms mentioned in section 6 there is a need for a testing environment.

There are several requirements to keep in mind when considering a testing environment.

1. The testing environment should be able to configure and run Crawljax.
2. The environment should have functionality to distribute the crawls or have other similar functionality to finish a test in a reasonable time span.
3. The database used by the environment should be able to deal with large amounts of data.
4. The environment should capture the DOM structures, besides all the regular results (e.g. states, edges, screenshots) of Crawljax.
5. The environment should collect the results in a centralized location, such as a database and/or server.

After doing a survey of available testing environments, which would fulfil the mentioned requirements above, we concluded that there is no testing environment available yet that meets the requirements. The crawling setup/environment mentioned in the paper of Nederlof et al. [13] is the most viable alternative. However, the software is specifically targeted at the goals of the relevant paper. It would require time to master and significant refactoring to make it comply to the requirements.
Therefore, the best option in our opinion is to build a new testing environment. The advantage of this option is that the entire testing environment can be custom tailored to the requirements, resulting in a lightweight application without any unnecessary functionality. The costs to build this small testing environment, would be more or less equal to refactoring the alternative option.

## 8 Implementation considerations

Knowing the requirements and the conceptual outline of the software, it is also important to consider the various choices that have to be made for the implementation. These aspects, such the choice of programming language, platform and additional components, should be carefully reviewed to ensure the optimal approach. In the following subsections we will discuss the possible trade-offs that need to be made in the fingerprinting algorithm and the testing environment.

### 8.1 Duplicate-detection algorithm

As there has not been much research on the effectiveness of the different duplicate-detection algorithms, it will require some explorative testing of the different algorithms to find the optimal one. We will begin with implementing simhash, as it was used successfully by the AJAX-enabled crawler of Peng et al. [14]
For the implementation of the duplicate-detection algorithm, there are only minor considerations possible. What is certain is that our implementation of the

algorithm will be implemented in or extending the Crawljax-platform Due to the nature of the internals of the Crawljax-core, the algorithm will also be implemented using Java.
Any components on which the algorithm depends will be managed using Maven[7], which is already used by Crawljax.
Lastly, a component will be needed to store and retrieve the fingerprints during a crawl-session. Crawljax is meant to crawl a single website at a time. Therefore only the fingerprints of the pages of the website currently being crawled need to be stored. The fingerprints are expected to be small enough to keep them in the memory. Otherwise, if the memory usage becomes an issue, a temporary lightweight database will be needed, for which the lightweight Database Management Systems (DBMSs) SQLite[8], Java DB (Derby)[9] or H2[10] would be excellent options, because these DBMSs do not require any servers or additional setup and are completely embedded in the application.

## 8.2 Testing Environment

The implementation of the testing environment as mentioned in section 7 provides, in contrast to the fingerprinting algorithm, more freedom in terms of the choice of platform, components and programming language.
However, in this case writing the software in Java is the best option. Crawljax and the custom plugins, that also need to be written, are already written in Java. Moreover, by using Maven in the testing environment, the management of the dependencies by both Crawljax and the relevant plugins are easier to manage. Additionally, useful features from the crawling environment by Nederlof et al. [13], also written in Java, can be reused.

Finally there is a choice to be made in terms of the database. The database does not need very advanced functionality, it mostly just needs to be capable of holding large datasets, including the screenshots.
The first decision that has to be made is whether a relational SQL database or a so called noSQL database will be used. The major advantage of noSQL over traditional databases is that it is fast and highly scalable.
Although noSQL has some theoretical performance advantages over relational SQL, we chose to use a relational database, due to our knowledge of SQL and the limited scope/timespan of our project. Therefore, we consider a noSQL database a nice-to-have feature.
When considering free relational databases, a choice has to be made between the two major DBMSs MySQL[11] and PostGreSQL [12]. Benchmarks[13] show that on large datasets PostGreSQL currently nearly always outperforms MySQL on

[7]http://maven.apache.org/
[8]http://www.sqlite.org/
[9]http://www.oracle.com/technetwork/java/javadb/
[10]http://www.h2database.com/
[11]http://www.mysql.com/
[12]http://www.postgresql.org/
[13]http://www.randombugs.com/linux/mysql-postgresql-benchmarks.html

large datasets. On the software-side, both DBMSs don't differ significantly. In the Java-environment for both DBMSs drivers are available. Thus, the testing environment will use a PostGreSQL DBMS.

## 9 Conclusion

Although there has not been much research on the implementation of duplication-detection algorithms in AJAX-enabled crawlers, there are a few well established algorithms used in static crawlers. Two of these currently most-used algorithms are Charkard's Simhash and Broder's shingling algorithm. Both algorithms are highly configurable in terms of the selection of features, which makes them excellent candidates to implement in the duplication-detection component of Crawljax. Therefore, we plan program and test implementations these algorithms in Crawljax. Because the algorithms' performance depends on the selection of features and weights given to features, each of the algorithms will have to be tested multiple times. Using this research, we hope to provide Crawljax with an efficient algorithm, which enhances the ability of Crawljax to detect and handle (near-)duplicate states.

## References

[1] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[3] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[4] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[5] Jeffrey Dean and Monika R Henzinger. Finding related pages in the world wide web. *Computer networks*, 31(11):1467–1479, 1999.

[6] Taher H Haveliwala, Aristides Gionis, Dan Klein, and Piotr Indyk. Evaluating strategies for similarity search on the web. In *Proceedings of the 11th international conference on World Wide Web*, pages 432–442. ACM, 2002.

[7] Monika Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM*

*SIGIR conference on Research and development in information retrieval*, pages 284–291. ACM, 2006.

[8] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[9] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150. ACM, 2007.

[10] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[11] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 122–134. IEEE, 2008.

[12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[13] Alex Nederlof, Ali Mesbah, and Arie van Deursen. Software engineering for the web: The state of the practice. 2014.

[14] Zhaomeng Peng, Nengqiang He, Chunxiao Jiang, Zhihua Li, Lei Xu, Yipeng Li, and Yong Ren. Graph-based ajax crawl: Mining data from rich internet applications. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, volume 3, pages 590–594. IEEE, 2012.

[15] William Pugh and Monika H Henzinger. Detecting duplicate and near-duplicate files, December 2 2003. US Patent 6,658,423.

[16] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. 2000.

[17] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011.

# Appendix B:
Plan of action

Erwin van Eyk Wilco van Leeuwen

Delft University of Technology
{E.D.C.vanEyk, W.J.vanLeeuwen}@student.tudelft.nl

May 2014

## 1 Preface

Crawljax is a JavaScript-enabled web crawler that can find all states of a web page, even if they are hidden by JavaScript actions instead of via regular links. However, this leads to a gigantic number of states with many duplicates. In this document we will explain how we will approach this problem and what agreements we have made with the client.

## 2 Summary

During this project the team will attempt to improve the detection of (near-)duplicate states in the open-source project Crawljax. The first objective is to construct a stable testing environment to easily and objectively benchmark the detection of duplicate states with a specific configuration. Secondly, our objective is to find and design a configuration or algorithm which provides optimal detection of duplicate states.
The client will provide us with feedback during the weekly or semi-weekly meetings on our progress. Additionally, the client will provide us with feedback online when we pull request the code to the main repository.

## 3 Introduction

There is not a optimal solution to detect near-duplicate states in terms of the content in the current version of Crawljax. This has two big disadvantages, 1) The crawloverview has way to much states, which makes the result of crawljax of less use and difficult to analyse with the human eye and 2) Crawljax needs to do more work than necessary, because it crawls a single state more times in stead of just noticing that this states is already been processed.

### 3.1 Agreement and Adjustment

During the first meeting with Alex and Arie, we discussed and agreed upon the aspects in this document. An additional email with an overview of the discussed aspects was send. This document has been composed using the verbal and non-verbal agreements.

### 3.2 Structure

In this document we will first elaborate the project assignment. Identifying the current situation, goal, deliverables and the associated constraints. Secondly, we will explain the agreed upon planning. After that, the project organization will be explained. Finally, the mechanisms concerning quality control will be identified.

# 4 Project Assigment

In this section we make more clear what the project consist of by explaining the basic concept of the project, agreements we made with our client, Alex, and the current status of the project we are going to work on.

## 4.1 Project Environment

Crawljax is a JavaScript-enabled web crawler that can find all states of a web page, even if they are hidden by JavaScript actions instead of via regular links. However, this leads to a gigantic number of states with many duplicates. This makes it necessary to use a proper duplicate detection to prevent Crawljax to crawl already crawled states. Currently, crawljax has a duplicate detection which use string comparison of a normalized DOM. [1] But the problem with this method is that minor or irrelevant changes can not be detected as duplicate states. We define irrelevant changes as changes that do not change the informative content of the state, what for the user is visiting the site. Irrelevant changes can be, for example, timestamp, advertisement or counter changes. Because of this changes, the duplicate detection will be called near-duplicate detection.

## 4.2 Project Goal

With a proper near-duplicate detection, it is possible for Crawljax to end up with a result of all states that contains different relevant content for the visitors, of the site that is crawled by Crawljax.

## 4.3 Project description

The duplicate detection of Crawljax can be improved significantly by implementing a fingerprinting algorithm used by many other crawlers. The project will consist of the following phases:

- Research several fingerprinting mechanisms used by other crawlers.
- Set-up a test environment for Crawljax the measures the number of (duplicate) states.
- Implement the new fingerprinting algorithm.
- Verify that it improves Crawljax's performance.

## 4.4 Deliverables

We will implement near-duplicate detection algorithms in Crawljax and prove by empirical experiments that this improves the performance of Crawljax.

## 4.5 Requirements and constraints

Programming will be done in Java. And the near-duplicate detection algorithm should be programmed with the same level of quality as the current implementation of Crawljax. We also have to work autonomously, but have every week a short meeting with Alex, the client.

[1] `http://bepsys.herokuapp.com/projects/view/17`

## 5 Problem Approach and Planning

The phases will be divided according to section 4.3.

**Phase I**
Research on basic crawlers, Crawljax and fingerprinting mechanisms and document the findings.

**Phase II**
Construction of a testing environment to compare the performance of Crawljax before and after the near-duplicate detection we will implement in Crawljax.

**Phase III**
Iteratively follow the next steps:

- Design/find a new fingerprinting algorithm
- Implementation of the fingerprinting algoritm
- Test the new version of Crawljax empirically
- Document the results of the test.

## 6 Project Organization

In this section we will discuss the constraints regarding the communication and other non-software aspects of the project. Due to the nature of the project and the organization behind Crawljax most of the aspects discussed will be similar to any other open-source project.

### 6.1 Organization

Due to the flexible and remote nature of the project, the team-members are both responsible for all types of tasks, including organizing, designing, implementing and communicating the progress with all parties.

### 6.2 Team

Both team-members are expected to be active on this project at least 40 hours per week. Besides unforeseen events, the team-members will be work on the project for at least 8 hours on the usual workdays. Additionally the members will be able to work during weekends and/or evenings if this is necessary because of deadlines.
Extensive knowledge of Java and complementary libaries and tools, such as Maven and Git, is a requirement. Additionally the client also likes to see knowledge about various high-level software concepts and the ability to work with large open-source software projects.

### 6.3 Administrative Procedures

The progress can be tracked by anyone looking into the open-source repositories used for this project. [2] [3] Branches and pull requests will be used to clarify the nature of specific progress. The team will also update any interested parties by mail.

[2] `https://github.com/erwinvaneyk/crawljax`
[3] `https://github.com/erwinvaneyk/crawljax-functional-testing-suite`

### 6.4 Financing

No financing is needed, because for the open-source project only free open-source libraries will be used. Any (database-)servers needed will be self-hosted or hosted using free services. The team-members will use their own equipment or equipment provided to them free of charge.

### 6.5 Reporting

Every week the team will update all concerned parties with a progress report by email. This email will contain a brief overview of what progress has been made, what issues where encountered and what will be planning of next week.
Alongside the email, the team will meet up with Alex to discuss the progress and issues in more detail. These meetings will be scheduled every one or two weeks, depending on the schedule of Alex.

### 6.6 Resources

The nature of this project is flexible and remote. Therefore the team will find their own workplaces. Additionally, the team-members will use their own equipment (computer, software ect.) and any equipment available at the TU Delft. If deemed necessary, Alex could provide a dedicated server for testing purposes.

## 7 Quality Control

The contact of the organization, Alex, will have the main responsiblity of quality control. Alex will comment on our code, online and/or during the meetings. He will also provide us with feedback and additional tips and tricks to improve the code. Furthermore, the guiding principle here for the quality of the code is that it has to have the same quality as the existing codebase.
To prevent any decrease in quality of the original codebase, the team will use git. During the project the team will code in a separate repository. When there is a supposedly finished product, a pull request will be issued to the main Crawljax-repository. Using this approach any concerned parties of Crawljax can review and comment on the code, thus ensuring the code quality. Additionally all code in a pull request has to be tested.

# Appendix C: Project description

## 1 Project description

Crawljax is a JavaScript-enabled web crawler invented at the TU-Delft and now maintained by the TU-Delft and the University of British Colombia. This crawler can find all states of a web page, even if they are hidden by JavaScript actions instead of via regular links. However, this leads to a gigantic number of states with many duplicates. To compare the states, we currently use string comparison of a normalized DOM. This could be improved significantly by implementing a fingerprinting algorithm used by many other crawlers. The project will consist of the following phases: - Research several fingerprinting mechanisms used by other crawlers. - Set-up a test environment for Crawljax the measures the number of (duplicate) states. - Implement the new fingerprinting algorithm. - Verify that it improves Crawljax's performance. You will learn: how crawlers really work, applying scientific methods, empirical analysis, working on a Open Source project.

## 2 Company description

Crawljax is an Open Source project that is maintained by TU-Delft and the University of British Colombia. Several big companies are backing the development including Intel, Fujitsu, SAP and Google. If you don't want to just implement some functionality for a company, but really want to do something challenging from both a development and intellectual perspective, this is your chance. Because the project is Open Source, this also looks great on you resume since all job seekers check your open source activity these days.

## 3 Auxiliary information

Programming will be done in Java. Requires knowledge of web applications. Only apply if you are unafraid of complicated problems, can work autonomously, and want to kick-ass. Read more about fingerprinting in this example paper: http://www.wwwconference.org/www2007/papers/paper215.pdf Project will be mentored by prof. Arie van Deursen and ir. Alex Nederlof, the CTO of Magnet.me and lead developer of the Crawljax project.

# Appendix D:<br>Blogpost

## 1 States receive fingerprints!

Tired spending most of the time crawling near-duplicate states? The solution has been added to the new release of Crawljax!

Up until now Crawljax had rather limited functionality for detecting duplicate states. This algorithm depends on a strict comparison between stripped DOMs. Only states with completely the same stripped DOM were considered to be duplicates. This caused the stateflow-graph to have lots of useless states that were near-duplicates.

In this release a solution will be provided for this problem, by allowing users to configure their own near-duplicate detection for Crawljax. Each state will have a fingerprint which represents the most essential parts of that state. This fingerprint will be used to identify a new found state as a unique one or as a near-duplicate of an already found state.

The near-duplicate detection system can be described as follows. Starting with the raw DOM, it will be converted to a stripped DOM using the existing DOM strippers. The strippers can be viewed as determining what parts of the content should be represented in the fingerprint. Afterwards, the stripped DOM is turned into small features. These features essentially describe how the content can be turned into fixed-length hashes. Finally the features are turned into one small-sized fingerprint. With the simple fingerprint-API it is very intuitive to compare fingerprints with each other.

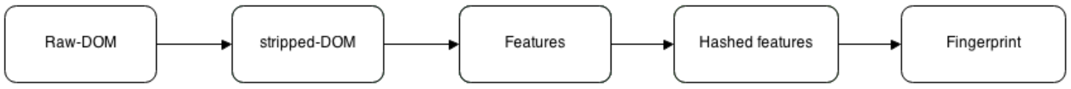


Figure 1: Generate fingerprint of a state

To check out this awesome new functionality either run the DuplicateDetectionExample in the examples. Or use the CLI to check it out by running the following command. java –jar crawljax-core.4.0.jar http://demo.crawljax.com/ example/ –useBroder There are several additional options available to tweak the functionality, which can be found in the documentation.

As a bonus, a so-called threshold-slider has been included along with the near-duplicate detection functionality. This cool little feature is basically a visualization of all the near-duplicate detection magic. By adjusting the slider the state-flow graph is shown under a different threshold.

Checkout some live examples of the slider here: http://erwinvaneyk.github.io/crawljax/.

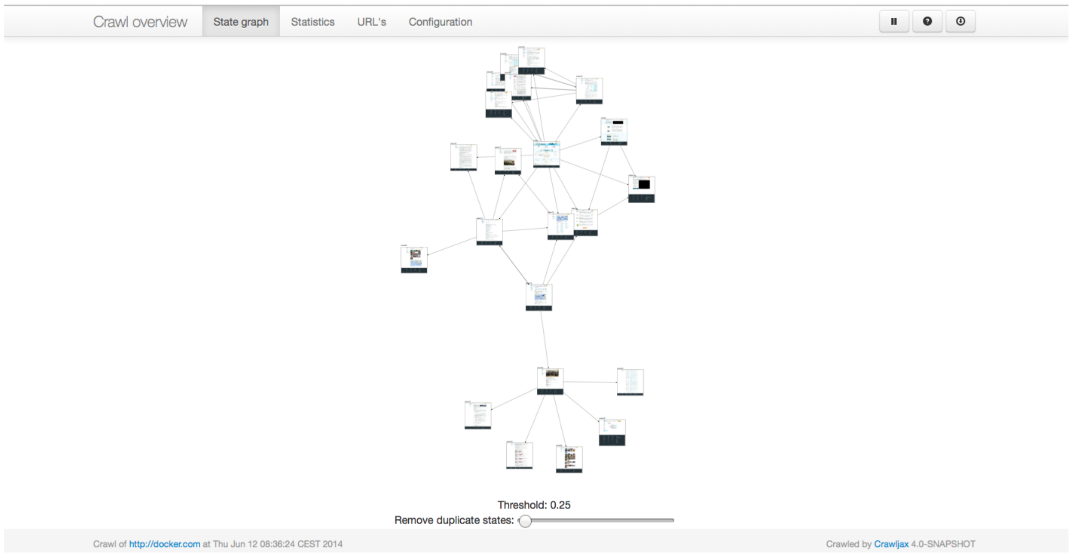


Figure 2: The state-flow graph under a threshold of 0.25

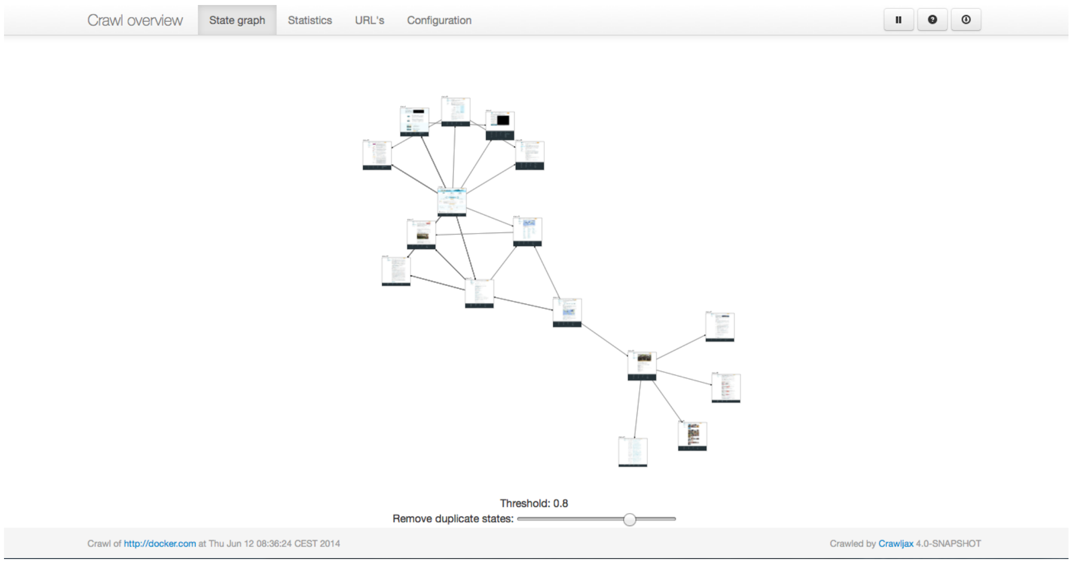
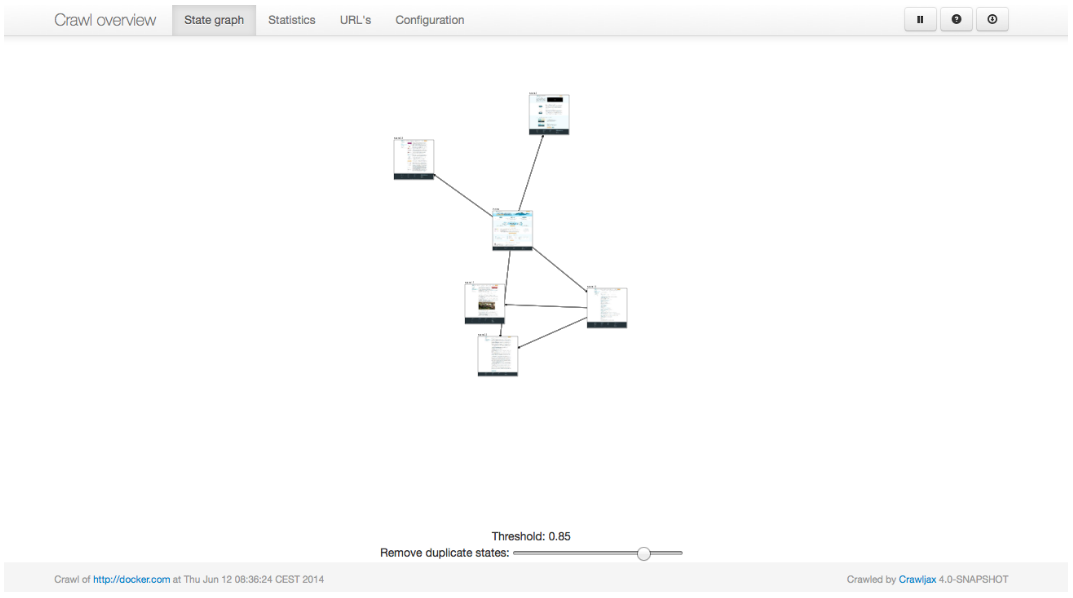

Crawl overview
State graph
Statistics
URL's
Configuration
Threshold: 0.8
Remove duplicate states:
Crawl of http://docker.com at Thu Jun 12 08:36:24 CEST 2014
Crawled by Crawljax 4.0-SNAPSHOT

Figure 3: The state-flow graph under a threshold of 0.8



Crawl of http://docker.com at Thu Jun 12 08:36:24 CEST 2014

Crawled by Crawljax 4.0-SNAPSHOT

Figure 4: The state-flow graph under a threshold of 0.85