Manfred Milchrahm

Improving Browser Bookmark Practicability

Bachelor's Thesis

to achieve the university degree of

Bachelor of Science

Bachelor's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Roman Kern

Knowledge Technologies Institute
Head: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt

Graz, September 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present bachelor's thesis.

21.9.2020

Date

Signature

Abstract

This present work elaborates on how a browser's bookmark functionality, a common tool to aid revisitation of web pages, can be improved concerning performance and user experience. After identifying and investigating issues arising with state-of-the-art approaches, solutions to that issues were elaborated and a browser extension for the Google Chrome browser was implemented based on the gathered insight. A special focus was put on developing novel functions that allow for incorporating temporal relations between bookmarks of a given bookmark collection as well as a feature that supports searching for bookmarked web pages by colour. Ten participants completed an evaluation of the implemented browser extension in order to investigate its performance and usability. The study showed that users familiarise quickly with the proposed novel functions and rated their ease of use and helpfulness positively. However, though the suggested functions were commented positively on by participants and showed advantages over traditional full-text search for special cases where some (temporal) context is required, full-text search extended by widespread functions like autocomplete suffice for most of the basic use cases.

Contents

List of Figures

1 Introduction

1.1 Motivation

According to studies (Tauscher and Greenberg 1997), (Cockburn and Bruce 2001), (Herder 2005), a major part of web browsing involves requesting web pages that have been visited before. As a consequence, the research topic of supporting web page revisitation emerged, leading to a number of browser extensions and other software following various approaches to enhance web page revisitation.

1.1.1 Bookmarks hold little relevant Information

Personalised (hierarchical) lists of URLs, sometimes called favourites or bookmarks, are one common way to save URLs for later use. Even though the bookmark feature is implemented in all current browsers, none of these browsers provide native tools to efficiently search and find previously stored bookmarks again to allow revisitation. Most browsers provide basic search functionality, which oftentimes does not deliver the desired results since there is very little information connected to a bookmark.

In this work we will discuss a method for improving the bookmark feature by adding more searchable information to bookmarks. The aforementioned problem concerns all current major browsers, so the approach suggested in the present work tries to provide a possible improvement that can be adapted for all browsers that allow developing extensions or plug-ins with access to a user's bookmarks. However, since each browser provides its own API and system architecture, a separate implementation is needed for each browser. Due to the popularity of the Google Chrome browser, the decision was made to make use of said browser's bookmark feature in the implementation proposed in this present work.

1 Introduction

The Google Chrome browser was developed by Google and released to the public in 2008[1]. As of January 2020, with 68.78% Google Chrome is the browser with the largest desktop market share, according to statcounter.com[2].

Google Chrome's adress bar is called omnibox, for it does not only allow to type in a website adress but functions as the main interface of the browser by providing additional, more advanced functionality. A user can type URLs or search queries into the omnibox. On user input, the omnibox displays search results gathered from sources like the user's default search engine, so-called rich results which are generated from sources like Wikipedia, as well as personalised results taken from a user's history and bookmarks. Although this may be convenient in some cases, due to a usability through simplicity approach, which, in essence, is hiding information from the user, this method is very intransparent with respect to how a result's relevance is rated and also which source the result was retrieved from. Also the number of search results presented to the user is limited. The main problem with omnibox search and current state-of-the-art solutions, as described later in Section 2.2, regarding searching for bookmarks is, though, that the browser often does not have much searchable or relevant information stored for a bookmark. These state-of-the-art approaches typically try to provide a way to add information like hierarchy or context to a users' bookmarks. However, the process of deriving and adding such information usually can not be automised completely. Hence, user effort is needed to be put into maintenance of the given bookmark collection. The present work seeks to reduce this effort.

Concerning searchable information, usually only a title or description of the bookmark and the time of when the bookmark was saved the first time is stored for each bookmark [3]. Google Chrome most likely uses the timestamp internally for rating the relevance of a result incorporating heuristics when searching via the omnibox. However, this approach is not transparent, as mentioned earlier, especially since the timestamp information is not indicated to the user, nor does the user know whether the source of a given omnibox result is their bookmark collection or browser history. Furthermore, the timestamp of the bookmark is not directly searchable in Google Chrome and also if it was, it would not be very userfriendly since a user would have to know the exact timestamp of the bookmark.

[1] https://en.wikipedia.org/wiki/Google_Chrome

[2] https://gs.statcounter.com/browser-market-share/desktop/worldwide/#monthly-201910-201910-bar

[3] https://developers.chrome.com/extensions/bookmarks#type-BookmarkTreeNode

The only information connected to a bookmark that users can utilise directly in their queries, is the title of the bookmark. In theory, the title of a bookmark can hold a lot of relevant information that can be exploited for search. In practice, however, users usually do not spend time to think about meaningful, descriptive or relevant bookmark titles, so oftentimes a default bookmark title is used, as Abrams, Baecker and Chignell found out (Abrams, Baecker and Chignell 2002). The default bookmark title suggested by the browser is simply taken from the HTML title meta tag of a website or some title tag of a PDF document, et cetera. When comparing typical HTML title tags to actual content, it is easy to recognise that there is much information available that is currently left unused. As Abrams, Baecker and Chignell put it, "bookmarks aren't great describers of the actual content [of the Web page]". They suggest this might be the case due to "traditional difficulties" when it comes to finding descriptive names for items. An example of this issue is shown in Figure 1.1.

1.1.2 The Effort of Maintenance

One common method of organising bookmarks is to create folders which the bookmarks can then be assigned to. These folders are usually organised in hierarchical structures. This method requires the user to explicitly create the folder structures as well as the bookmark to folder assignments which tends to get unhandy for bigger bookmark collections. As explained in more detail in Section 2.2.3, there are approaches to reduce this maintenance effort, for example through automatic categorisation. Still, these tools do not solve the following problem: when searching for a certain bookmark, the user has to figure out to which automatically created folder the bookmark has been assigned to (Do and Ruddle 2017) and navigate through a potentially large number of bookmark folders. This task becomes even more of a problem as these tools tend to have difficulties classifiying bookmarks correctly. For example, HyperBK (Staff and Bugeja 2007), presented in Section 2.2.3 was not able to correctly classify 39% of given bookmarks. Thus, our approach aims to get rid of having to assign bookmarks to certain folders and putting much effort into folder maintenance by providing the user with a simpler search interface.

1 Introduction

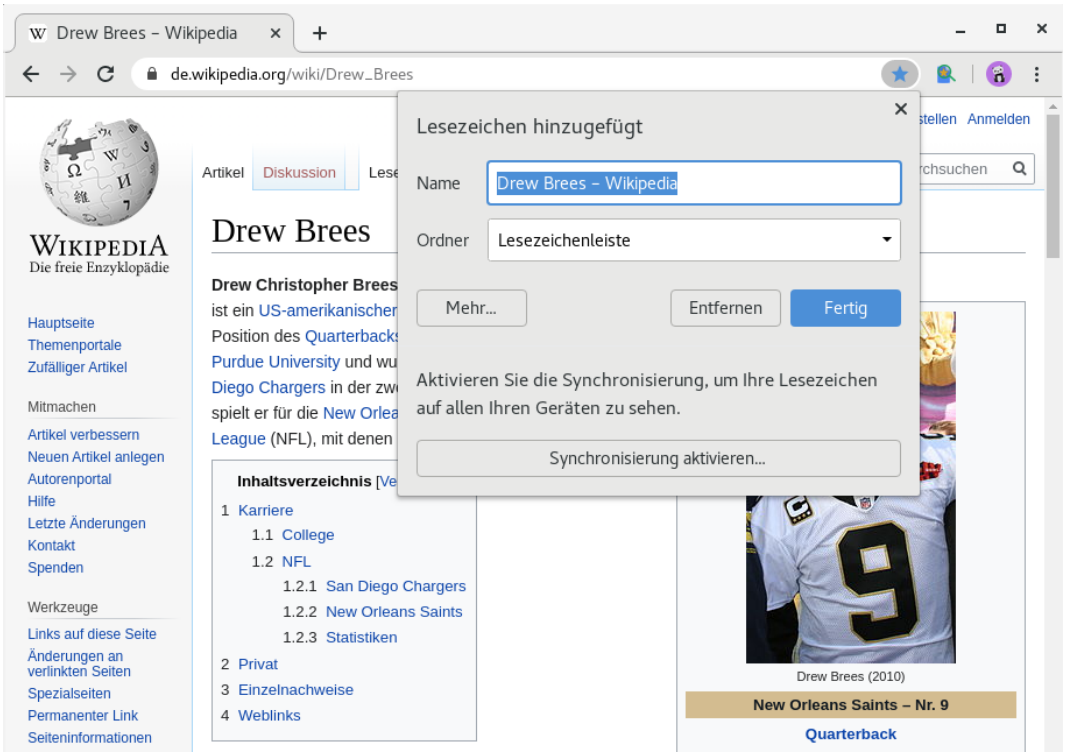

Figure 1.1: The Wikipedia page of Drew Brees: on creating a bookmark, the browser suggests the web page's HTML title tag "Drew Brees - Wikipedia" as the bookmark title, when there is much more relevant information that could be used. For example, "Quarterback", "Football", "Chargers", "New Orleans" and "Saints" would provide relevant searchable terms. Imagine now searching your bookmarks for an NFL player who's name you just forgot.

1.1.3 Adding Information effortlessly

Storing little information and not extracting any content made sense in the early days of the internet, due to little computer performance. Nowadays relatively big and fast memory allows for search indices being held locally on a client computer. Moreover, today's internet infrastructure allows for parallelising the task of fetching websites in order to process them and add to a search index. As a result, bookmark collections with large amounts of documents can be fetched and indexed relatively fast even on a client device.

In this present work, we make use of the content of bookmarked documents by building a search index out of this content in order to add searchable information to bookmarks without having to put effort into maintenance. Furthermore, in an approach to provide for more efficiency and thus a better user experience, a search interface that allows for advanced search queries is discussed. Finally, the approach is evaluated and the results are discussed in Chapter 4.

A qualitative user evaluation was conducted to measure (a) if, and if yes, how the plugin helps improving users' confidence in having found the desired document, and (b) if the plugin helps raising a users' willingness to use bookmarks more frequently, or if the user did not use bookmarks before, the willingness to use bookmarks at all for website revisitation purposes.

1.2 Main Challenges

In order to perform indexing of browser bookmark data there are typical Natural Language Processing (NLP) tasks which have to be considered. Especially when keeping localisation and multilingual environments in mind, NLP tasks may pose a difficult problem to solve. An example of such tasks are Tokenisation and Stemming, which are being dealt with in more detail in Chapter 2.

1.2.1 Memorising Websites

Besides aforementioned NLP considerations, the main task of this work is to implement a query system that takes advantage of how a human user memorises

bookmarked websites.

In order to understand how users formulate queries when searching for a specific bookmark, it is important to take into account that users sometimes do not memorise the exact name of a bookmark. In this work we hypothesise this is especially the case when bookmarking websites which have not been accessed directly, but through some hyperlink, for example after querying a search engine. For example, a user queries a search engine looking for a used Ferrari and finds one on some used car website and bookmarks the page. After two weeks the user wants to access the bookmarked website again. It is much more likely that the user will query their bookmarks collection for something like "used car" or even "used ferrari" rather than the name of the used car website.

Another hypothesis we propose is, that users think in so-called "temporal episodes" when creating bookmarks. For example, after creating a bookmark for a website presenting some front end development tools, the user, who is a web developer, also created a bookmark for some JavaScript framework. With the help of a temporal search feature, the user now could query for "*JavaScript* that has been bookmarked shortly after *front end*" in order to improve search efficiency, instead of just searching for "Javascript". Of course this approach requires the user to explicitly bookmark at least two interrelated web pages within the same browser session. It has to be evaluated if such behaviour corresponds to reality, or if the requirement to manually create bookmarks poses a problem to this approach. This problem could be solved by using browser history, which is recorded automatically, instead of manually created bookmarks. It is worth noting that some browsers already support full-text search for browser history, but not for bookmarks. However, given the fact that most major browsers do not keep an unlimited amount of browser history records due to performance reasons (Sousa, Pereira and Martins 2012), using browser history poses no solution to long term revisitation.

We also hypothesise that users are likely to memorise dominant colours of a website. While there have numerous studies been conducted on the effects of colour in the context of memorisation in the fields of e-commerce (Pelet and Papadopoulou 2012) and e-learning (Pelet and Papadopoulou 2011), as of now the question remains whether colour can be perceived in such a manner that allows for remembering the colour or shade and formulating good information retrieval queries after some given time. A given colour does not necessarily have to be memorised consciously,

instead it is relevant if a colour can be recalled to memory even if it has been perceived unconsciously at the previous visit of the concerned web page.

Therefore it is important to find a way to efficiently retrieve, extract and index relevant information like textual content, colour and temporal context from a given bookmarked web page, so that this information can be made searchable for the user. How textual content can be extracted and indexed is shown in Chapters 2 and 3.2, where a major challenge is the resulting index size for potentially large bookmark collections.

Temporal context can be extracted relatively easyily by making use of the bookmarks' timestamps. In contrast, extracting and processing colours is a rather difficult task. On the extraction side, there is the issue of determining which colour of a website really is prominent to the user. For example, if a website is made up of a large proportion of white background and just a few coloured, say, red, accents that catch the user's eye, the user, if asked, would probably not state the website to be "white", but instead remember the red accents, even though the mathematical proportions of the colours would suggest otherwise. Another problem that arises here, is that different users can have a different perception of one specific given colour. A very prominent example of this effect is the photograph of a dress, which is perceived by some people as white and gold, whereas others see the dress as blue and black (Chetverikov and Ivanchei 2016). However, the effect does not necessarily have to be as drastic. Take, for example, colours with very low saturation. Some people still can recognise some colour, but others are less sensitive to saturation and therefore just see grey. Naturally, this poses a problem for information retrieval, because we do not know for sure by which term a user will refer to a given colour.

As this work focuses on the question if bookmarks can be used more efficiently and not on content extraction, instead of writing our own content extractor, the decision was made to use a state-of-the-art third party content extractor called Tika, which is presented in Section 2.1.2. However, the incorporation of Tika in a browser extension poses another challenge, since extensions are typically sandboxed and therefore do not allow the use of the operating system's JRE. Consequently, since Tika is implemented in Java and depends on being run within a JRE, it can not be executed directly within such a sandboxed extension. As a compromise, in this present work Tika is being run on an external Node.js server as described in Section 3.2.2.

2 Background

2.1 Related Work

2.1.1 Node.js

Node.js is an open-source JavaScript runtime environment that allows for running very resource-saving webservers. JavaScript's event-driven architecture enables the webserver to have a large number of open connections due to low memory usage. Node.js's basic functionality can be extended with so-called modules, which can be installed, for example, via Node's package manager npm. How such modules have been incorporated in this present work's content extraction server is described in Section 3.2.2.

2.1.2 Apache Tika

The aim of this work is for each of a user's bookmarks to gather a lot more information than current solutions do. Current solutions, like the default in-browser bookmark functionality and even third-party extensions only use a small portion of available information and big parts of relevant information gets lost, thus obviously can not be employed for search. Our approach parses each of a user's bookmarked documents and stores the whole extracted content to be used for later search. We do so by making use of Apache Tika[1], which is a toolkit to parse documents by detecting and extracting data from various document types like HTML, PDF or office documents. The information retrieved by Apache Tika can then be used to build a search index.

[1] https://tika.apache.org

Originally, Apache Tika was written in Java. For this present work the decision was made to utilise a Node.js port of Tika by Matthew Caruana Galizia[2], which can be installed via npm. This module makes use of the node-java bridge[3], in order to provide for a JRE that is a requirement for Tika.

2.1.3 Indexing

The purpose of a search index is to provide for fast querying. There are various approaches for index design trying to optimise factors like query time or index size by making use of various data structures like trees, matrices or inverted indices. In contrast to simply storing every document as-is and searching all of them sequentially on each query, indexing is faster for potentially large amounts of documents. However, usually there is a trade-off between query time and build-up time.

2.1.4 Precision and Recall

Precision and recall both are typical measures of relevance in information retrieval. They are defined in terms of a set of retrieved documents and a set of relevant documents[4], where precision is defined as

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

and indicates how many of the retrieved documents are relevant. On the other hand, recall is defined as

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|},$$

which is the number of retrieved relevant documents out of all relevant documents.

[2] https://github.com/ICIJ/node-tika

[3] https://github.com/joeferner/node-java

[4] https://en.wikipedia.org/wiki/Precision_and_recall#Definition_(information_retrieval_context)

2.1.5 Tokenisation

Tokenisation describes the process of splitting text into smaller entities, called tokens, each of which having semantic value. While this task is rather straightforward for languages like German or English, there are languages for which tokenisation is not as trivial. This is the case for languages like Chinese, where word boundaries are not indicated by whitespace characters and therefore are ambiguous.

2.1.6 Stemming

Stemming is a process that aims to reduce tokens to their common word stem, or root, in order to group words with similar meaning. It is not necessarily required to find the actual morphological root, it is oftentimes sufficient that semantically similar words map to the same stem. This can be beneficial to rating documents, because the recall is increased. High recall essentially means, as explained earlier in Section 2.1.4, that most of the relevant documents are retrieved. For example, a document containing the word "Flowers" is most likely relevant to a user searching for "Flower". Furthermore, stemming reduces the total number of tokens in the index, thus results in better performance.

There are different types of stemming algorithms, or stemmers, which can be grouped into a number of categories. For example, there are truncating algorithms, like the Porter Stemmer (Porter 1980) used in this present work, or statistical approaches like n-grams. While truncating algorithms, as the name suggests, remove suffixes or sometimes prefixes in order to group words by their root word, the n-gram algorithm generates substrings of length n extracted from consecutive characters of a word. As a result, n-gram algorithms produce $\mathcal{O}(n)$ n-grams for each word. The idea of the n-gram algorithm is that syntactically similar words have a high number of common n-grams. According to Damashek (Damashek 1995), the n-gram approach is superior to truncating algorithms in a multilingual environment since statistical approaches are language-independent, which does not hold true for truncating algorithms as explained below. Nonetheless, for the time being, we stick with the default stemming algorithm of the indexing framework used in this present work, which is one of the most commonly used truncating stemming algorithms, the Porter stemmer, which allows for smaller index sizes. The Porter stemmer is a rule-based stemming algorithm, which means a given set of

rules is applied on each token, until a certain minimum number of syllables remain. For example, OSCILLATORS is stripped to OSCILLATOR, then to OSCILLATE, then to OSCILL, and then to OSCIL. This example illustrates that the stem a token is reduced to does not necessarily have to be a valid word, but related words should produce the same stem, which usually is sufficient.

It is important to keep in mind that a rule-based stemming algorithm, due to its underlying rules, is language-dependent. While the original Porter stemmer was implemented for the English language, there are versions for other languages like German, French and Russian available as well by now. It is worth noting that stemmers for languages like Arabic and Hebrew are still regarded as a difficult research topic. This has to be taken into account when implementing support for different localisations or multilingual bookmark collections, which is not part of this work though.

2.1.7 TF-IDF

To rate the relevance of a specific document for a given query, various statistics can be used. One of the best performing and most commonly used ones is term frequency-inverse document frequency (TF-IDF), which is used by the Lunr framework that is presented in Section 2.1.8.

Term frequency describes how often a term appears in a document. In the simplest case this value is just the raw number of occurences of the specific term.

Since there are words that are very common, their appearance in a document does not give reliable information about the relevance of the document. For example, the occurence of the very common word “the” is likely to boost the score of a document while diminishing the emphasis put on words that are more distinctive. In order to compensate for this falsely emphasised terms, term frequency is multiplied by an inverse document frequency factor, which is a measure of how common a word is across the whole document collection, hence how significant an occurence of this word is. The specific computation of this value by Lunr is described in the following section.

2.1.8 Lunr.js

Lunr.js, or simply Lunr[5], is an open-source full-text search framework written in JavaScript.

For determining similarity between and ranking of documents, Lunr uses a vector space model. In this model, for each document a multi-dimensional vector is computed. In fact, since a document can be composed of multiple fields, there can be multiple vectors describing the document, one vector for each field. Each of a vector's dimensions corresponds to a term that occurs in the regarding document field and a weighting score is assigned to each dimension, or term, respectively. These scores are computed by using an implementation of the BM25 algorithm (Robertson et al. 1995), which is a state-of-the-art TF-IDF algorithm. To be precise, an improvement of BM25, referred to as BM25F (Zaragoza et al. 2004), is being employed. BM25F allows for better performance across multiple weighted fields. In Lunr's implementation[6], given a query term q, the score of a document's field f is computed as

$$\text{score}(f, q) = \text{IDF}(q) \cdot \frac{\text{tf}(q, f) \cdot (k_1 + 1)}{\text{tf}(q, f) + k_1 \cdot \left(1 - b + b \cdot \frac{|f|}{\text{avgfl}}\right)},$$

where $\text{tf}(q, f)$ is the term frequency of the term q within field f, $|f|$ is the field length in number of terms, avgfl is the average field length of all documents in the collection and k_1 and b are free parameters with default values 1.2 and 0.75 respectively, that can be used for advanced optimisation.

The IDF weight is computed as

$$\text{IDF}(q) = \log\left(1 + \left|\frac{N - n(q) + 0.5}{n(q) + 0.5}\right|\right),$$

where $N = |D|$ is the total number of documents in the collection, and $n(q) = |\{d \in D : q \in d\}|$, that is the number of documents d containing term q[7].

The similarity between a document and a query is then computed by computing the dot product of the corresponding vectors.

[5] https://lunrjs.com/
[6] https://github.com/olivernn/lunr.js/blob/master/lunr.js#L2576
[7] https://github.com/olivernn/lunr.js/blob/master/lunr.js#L319

2.2 State of the Art

2.2.1 Keyword Search and Full-text Search

Keyword search solutions, as the name suggests, annotate bookmarks with keywords which then can be searched for. These keywords are either retrieved automatically or have to be defined by the user for each bookmark. Sometimes these two approaches are combined by suggesting keywords which then can be edited further by the user. However, keyword search has the obvious disadvantage that relevant content can be missed when choosing the wrong keywords.

In contrast, there already exist full-text search applications, that allow for retrieving and indexing the complete content of a given web page. Examples for such browser extensions that have been implemented for the Google Chrome browser are Falcon[8] and WorldBrain's Memex[9]. One issue regarding these browser extensions that should be considered, though, is that these extensions do require the user to visit a web page after the extension has been installed in order to index the contents of the concerned page. This is due to the extension parsing the DOM of the web page instead of requesting the page autonomously and independently from the user. Consequently, two drawbacks arise, the first one being that some of these extensions can not index web pages that have been bookmarked or visited before the installation of the extension, which obviously can lead to a user being unable to find web pages that have not been visited recently. Data protection issues may pose another problem, since these extensions are allowed to access the complete DOM of a viewed web page. To be clear, this means that such an extension has access to any page the user is able to access. For example, the content of web pages that are password-protected is exposed after the user logs in to such pages, hence sensible data like bank account sites might be exposed. That said, these extensions usually do provide the possibility to exclude certain pages from being indexed, but considering real world user behaviour, the question remains if this opt-out approach is a good fit from a data privacy perspective.

Furthermore, although Memex allows for filtering by date range and even formulating those date ranges in natural language, a user is required to remember the

[8] https://chrome.google.com/webstore/detail/falcon/mmifbbohghecjloeklpbinkjpbplfalb
[9] https://chrome.google.com/webstore/detail/worldbrains-memex/abkfbakhjpmblaafnpgjppbmioombali

(approximate) date the desired web page was visited or bookmarked. While this might be less of an issue when searching for recently added records, it might become problematic when trying to obtain results from long term history. On the other hand, the approach employed in the present work tries to provide for narrowing down search results by allowing for formulating a temporal relation between two search terms instead of the temporal context of just a single search query. That is, for example, searching for records that (a) match term A, and (b) have been viewed or bookmarked shortly after results that match term B. An implementation of such a feature is shown in Section 3.2.1. In Chapter 4 this method has been evaluated.

2.2.2 Tagging

There exists a number of browser extensions, for example Bookmark tags[10] that provide a tagging feature for a user's bookmarks. Essentially, this is not much of a difference to keyword search, but tags usually are represented in a specific visual manner. Similar to keywords, a single tag also can be composed of one or more words. Such grouping can be an advantage over just writing all keywords into the bookmark title.

Besides the usual maintenance effort issue, keywords and tags have following typical problems as well.

Synonyms

Using tags that have same or similar meaning is one typical information retrieval problem. For example, when a user tags some bookmarks as "football" and others as "soccer", querying for "football" does not yield all bookmarks that concern a "a sport played by two teams of 11 players, who try to kick a round ball into their opponents' goal"[11].

[10] https://chrome.google.com/webstore/detail/bookmark-tags/edpeidcfjfmepdgdjnodefckgdjbigem

[11] https://www.ldoceonline.com/dictionary/soccer

Homonyms

A homonym is a word that has different meanings. For example, the word bow can refer to a long wooden stick with horse hair that is used to play certain string instruments or a weapon to shoot projectiles with. As opposed to synonyms where queries can miss a subset of relevant results, homonyms can lead to retrieving too many results. That is, obtaining results that contain the queried term but are semantically irrelevant.

Level of Abstraction

Using tags with different levels ob abstraction can have the same consequences as using synonyms. For example, a user might tag some web pages by the term “beer” in general, but differentiate between lager, ale and stout for some other websites. The same user might even tag some web pages as “Guinness” specifically, that is, refer to the web page by the brand name instead of the type of beer.

Typos and different Ways to write Keywords

While different usages of plural and singular, for example sometimes tagging “pints” and some other time using the singular “pint”, can be addressed by some kind of stemming as explained in Section 2.1.6, typos or different notation, for example “PintOfGuinness” versus “pint_of_guinness” versus “Pint of Guinness”, maybe pose an issue that can not be tackled by (rule-based) stemming as easily.

In order to address the aforementioned issues, Social Semantic Bookmarking (Braun, Zacharias and Happel 2008) tries to add semantic value by providing context through using ontologies, but again the user is required to put maintenance effort into such ontologies and keeping semantic relationships up-to-date.

2.2.3 Automatic Categorisation

HyperBK

HyperBK is an extension for the Firefox browser that tries to classify a given web page into a bookmark folder, where the title of a bookmark folder is made up of the keywords of the bookmarks the folder contains. One disadvantage of this approach is that only a fraction of the keywords extracted from a web page is actually being used and stored. Another issue is that a user still has to figure out to which folder a bookmarked web page she wants to revisit has been classified to and navigate through a potentially complex folder hierarchy in order to find it.

HiBo

Another approach similar to HyperBK was proposed by Kokosis et al. HiBo (Kokosis et al. 2005) also employs techniques to categorise and organise bookmarks. In contrast to HyperBK, HiBo takes a predefined set of topical hierarchies as input and classifies a given bookmark collection automatically. Thus, HiBo has the same drawback as HyperBK, as the user has to figure out to which topical category the bookmark has been assigned to.

3 Method

3.1 Concepts

3.1.1 Model–View–Controller

Model-view-controller (MVC) is a software design pattern that is mainly used for developing front-end graphical user interfaces (GUI). The aim of this pattern is to improve maintenance, flexibility and reusability of code. This is done by separating program logic into the three components model, view and controller. Each component is responsible for a certain part of the program logic, but communicates with the other components, either directly or via an observer pattern as illustrated in Figure 3.1.

The model holds the program data. If any of the data is updated, which usually is done by the controller, the view is notified via the observer pattern.

The view is responsible for the presentation of information to the user. This is done by fetching the data from the model. On user interaction the view notifies the controller to handle the input.

The controller handles view and model. It gets informed about user interaction by the view, processes the information of that user interaction and updates view and model accordingly.

3.1.2 Observer Pattern

The observer pattern is commonly used in software design for implementing event handling, especially in systems where data is not ready for use at startup, but instead comes in at some unpredictable time. An event maintains a list of observer

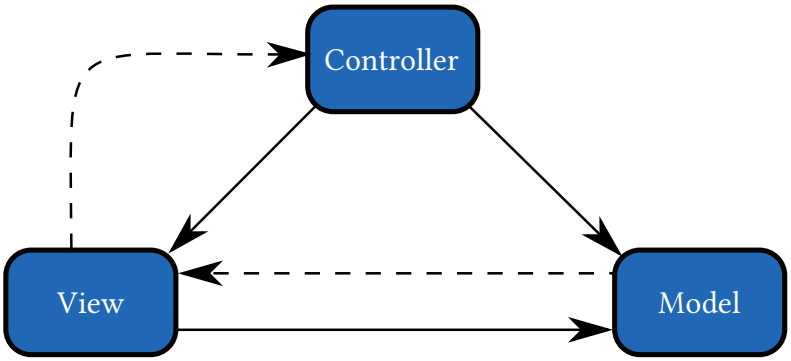

Figure 3.1: MVC Architecture: Solid connections represent direct associations between components. Dashed lines indicate connections via an observer pattern, where the direction of the arrow represents informing an observer about the occurrence of an event.

objects. An arbitrary number of observers can register to an event. An event could be some user interaction or an update of the data in the model. When such an event occurs, all registered observers are notified that this event happened. Usually a certain method of the observer object is being called upon notification.

3.1.3 AJAX

Basically, JavaScript scripts run in a single-threaded environment where functions are added to a so-called event loop. JavaScript then processes each of these functions sequentially. There are methods to defer function calls for a certain amount of time, but these methods just defer the point in time where the function call is added to the event loop. When the deferred function call is added to the event loop, the function is still being processed in order. However, JavaScript allows for non-blocking execution of time-consuming procedures by providing mechanisms to call functions asynchronously. This concept is called Asynchronous JavaScript and XML (AJAX) and usually depends on a so-called XMLHttpRequest (XHR) object, which allows to send a server request and wait for the response in a non-blocking manner. On response the XHR object is notified and then can execute a so-called callback method. This means, for example, that user interface interactions can be handled while waiting for a response of a server request.

3.1.4 Parallel vs. non-blocking Javascript

As described in the previous chapter, AJAX can be used to make asynchronous requests. However, AJAX does not solve the issue of computationally expensive script functions that do not rely on a server call, but are executed within the script locally. Such functions de facto do exist, for example when processing large amounts of information locally. Due to the single-threaded nature of JavaScript, such processing will block even if it is executed asynchronously. Web workers address this issue. Specifications for this programming interface are currently being developed by the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG), however, the feature is already supported by most current browsers. A web worker essentially is a JavaScript script that runs in a different thread than, and therefore is independent of, the main script. However, it usually is necessary that the web worker communicates with the main script. This task is achieved via message passing. Listings 3.1 and 3.2, inspired by WHATWG's living standard specifications[1], illustrate how this can be done. The main script is as follows:

Listing 3.1: Creating a web worker in the main script and listening for messages sent by this worker.

```
var worker = new Worker('worker.js');
worker.onmessage = function (event) {
  var result = event.data;
};
```

Listing 3.2: The web worker (worker.js) uses the postMessage() method to communicate the result to the main script.

```
let rand = Math.random()
postMessage(n);
```

How such webworkers have been utilised in this present work is described in 3.2.1.

[1] https://html.spec.whatwg.org/multipage/workers.html

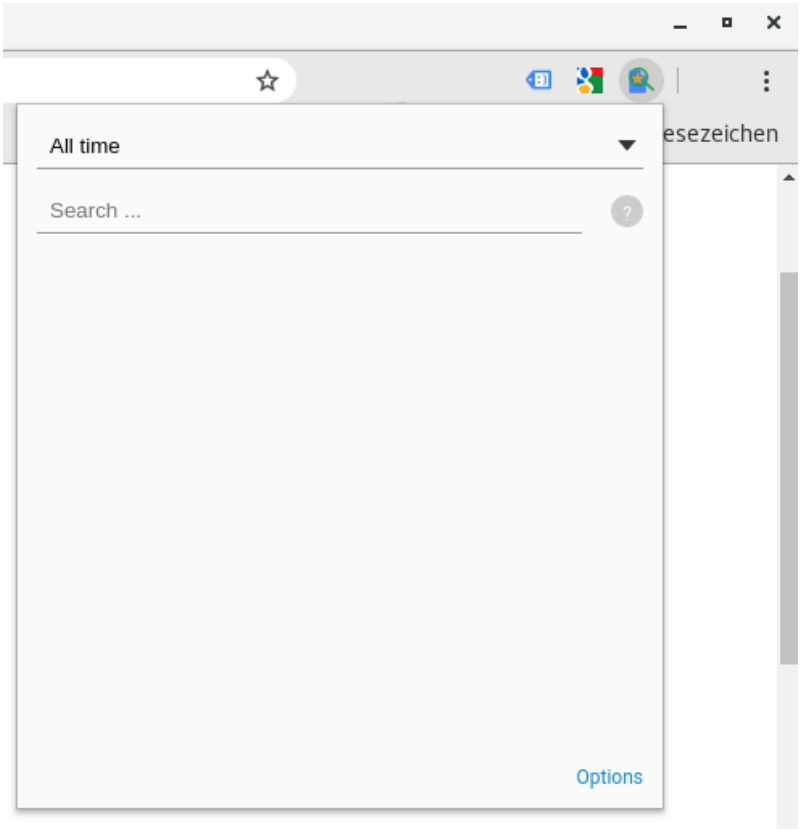

Figure 3.2: By clicking on the extension's icon the popup is loaded.

3.2 Implementation

3.2.1 The Browser Extension

A Google Chrome browser extension is a set of HTML, CSS, JavaScript and image files. The manifest file `manifest.json` informs the browser about important files and permissions needed by the extension. Another required file is the icon image file titled `icon.png`, that can be used by the browser as a button to trigger the popup that serves as the user interface as shown in Figure 3.2.

Popup

A popup page essentially is a regular HTML page containing all UI markup. In this present work basic CSS definitions have been added inline to the popup file (titled `popup.html`) itself, whereas JavaScript logic is being loaded from separate files as required by Chrome's Content Security Policy[2]. The JavaScript files being requested are shown in Listing 3.3. The file `popup.js` contains the script to create and initialise the MVC and observer instances. It is worth noting that the order of the scripts matters since there are mutual dependencies between the files. Also note that these scripts are unloaded upon closing of the popup, for example by performing a mouseclick outside of the popup page.

Listing 3.3: JavaScript files requested at the end of the popup.html file. Model, view and controller are initialised by the popup.js script. The order of the files is important due to dependencies between them.

```
<script src="event.js"></script>
<script src="model.js"></script>
<script src="view.js"></script>
<script src="controller.js"></script>
<script src="popup.js"></script>
```

Background Script

Background scripts usually are not visible to the user, but are loaded and unloaded as they are needed. A background script and its context, that means all its functions and variables, can be invoked by any other view, for example a popup. Background scripts contain event listeners that are crucial to the extension. In this present work, the background script contained in the `eventPage.js` file has event listeners for adding, updating or removing a bookmark as shown in Listing 3.5. This is also the place where the index creation is triggered by the `chrome.runtime.onInstalled` event. However, the index creation can also be initiated by the controller by requesting the background page as shown in Listing 3.4

Listing 3.4: The controller can request the background script and instruct it to start index creation.

```
chrome.runtime.getBackgroundPage(function(bgPage) {
```

[2] https://developer.chrome.com/extensions/contentSecurityPolicy

```
  //provide user with feedback to indexing progress
  _this._view.showProgress('block')
   //used to allow just one indexing progress at a time
  _this.saveIndexingStatus(1)
  bgPage.startIndexCreation()
})
```

In contrast to scripts loaded in a popup as described in Section 3.2.1, a background script is not unloaded as long as it executes a task. Note that background scripts also will not be unloaded as long as any messaging port or visible view, like a popup, is open. That means, since indexing is done in a background script, index creation and manipulation does not depend on any popup but happens in the background hidden from the user. The only limitation in terms of background script lifetime regards the closing of the browser window. An indexing process can be interrupted by closing the browser window either intentionally or not, for example when the extension or browser crashes, which possibly leads to data loss.

Listing 3.5: A background script listening for events the extension relies on. The addBookmark(document, update=0) respectively removeBookmark(id) handle the request for updating the index.

```
chrome.bookmarks.onCreated.addListener(function(id, bookmark) {
  if(bookmark.url){    //if there's no URL, it's a folder
      var document = {
        "bookmarkID": bookmark.id,
        "bookmarkURL": bookmark.url,
        "bookmarkTitle": bookmark.title,
        "bookmarkContent": "",
        "bookmarkMeta": "",
        "bookmarkColors": ""
      }
      addBookmark(document)
  }
})

chrome.bookmarks.onRemoved.addListener(function(id, removeInfo) {
  removeBookmark(id)
})
```

Message Passing

Oftentimes some kind of communication between the different parts of the extension is required. For example, it is beneficial to the user experience when the background script informs a popup about the progress of the indexing process. Since background and popup script variables reside in separate scopes, there is no direct way of sharing any information. However, `chrome.runtime` provides a mechanism that allows for such communication. This API is used in this present work as shown in Listings 3.6 and 3.7. While the background script sends progress notifications, any controller is listening to the `onMessage` event and handles incoming messages accordingly, for example by loading a newly created index or by passing the information on to the view which again renders the new information for the user.

Listing 3.6: The background page uses chrome.runtime's sendMessage() method to communicate its progress to every content or popup script listening.

```
  chrome.runtime.sendMessage({
     message: "THIS IS BACKGROUNDPAGE TALKING: progress notification.",
     docs: numDocs,
     processed: docsProcessed
  })
```

Listing 3.7: The controller listens for the onMessage event and handles messages from the background script accordingly.

```
chrome.runtime.onMessage.addListener(
  function(request, sender, sendResponse) {
    if(request.message.includes("index has been saved")) {
      ... //clean up
      _this.loadIndexFromStorage() //load newly saved index
    }
    if(request.message.includes("index not saved")) {
      ... //error handling
    }
    if(request.message.includes("progress notification")) {
      //give feedback to the user
      _this._view.setProgress( request.processed / request.docs * 100)
    }
  }
)
```

Chrome Storage

Chrome storage[3] is an API that provides storing and retrieving functionality. It can be used by an extension to persist user data. That way, user data does not get lost when the browser is being closed. For the present work, this API has been utilised to store the index that was created from a user's bookmarks, so that the index has not to be rebuilt upon browser restart. Furthermore, any extension settings can be persisted. It is worth noting that the API provides a `chrome.storage.local` as well as a `chrome.storage.sync` area. Though the sync area, as the name suggests, can be used to store information that should be synchronised along different devices, since this storage area is limited to just a few kilobytes, it does not serve the purpose of storing potentially large indices very well. The local storage area's storage limits are a lot less restrictive, large bookmark collections' indices still can exceed these limits as well. However, in contrast to the sync storage area, the storage limits of the local storage area can be bypassed by setting the `unlimitedStorage` permission in the extension's manifest file. This option was used in this present work to allow for large index sizes. Listings 3.8 and 3.9 show how the controller uses the storage API for persisting the index within the `lunrjsindex` item.

Listing 3.8: The controller's function to load the index from local storage. In the callback function it is checked if an index has been created yet. If not, instruct the background page to create one.

```
loadIndexFromStorage: function() {
  var _this = this
  chrome.storage.local.get('lunrjsindex', function(result) {
    if(result['lunrjsindex'] !== undefined) {
      //send idx to worker for autocomplete function
      _this._acWorker.postMessage({"idx": result['lunrjsindex']});

      var idx = lunrMutable.Index.load(
        JSON.parse(result['lunrjsindex'])
      )
      _this._model.setIdx(idx)
      ...
    }
    else {  //if index has not been created, do it now
      chrome.runtime.getBackgroundPage(function(bgPage) {
          ...
          bgPage.startIndexCreation()
```

[3] https://developer.chrome.com/apps/storage

```
      });
    }
  });
}
```

Listing 3.9: When the background page (eventPage.js) completes creating an index, it stores the serialised index to local storage. Upon successful storing notify the controller about the newly available index.

```
function saveIndexToStorage(serialisedIdx) {
  chrome.storage.local.set({'lunrjsindex': serialisedIdx},
    function() {
      ... //clean up progress status variables

      //notify the controller that the index is available now
      chrome.runtime.sendMessage(
        {
        message:
        "THIS IS BACKGROUNDPAGE TALKING: index has been saved."
        }
      )
    });
}
```

As mentioned on the API description webpage, Chrome storage is not intended to store confidential user information since the storage area lacks encryption.

Installation

Generally, to publish a Google Chrome extension, the extension is "distributed through the Chrome Developer Dashboard and published to the Chrome Web Store"[4], from where it can then be downloaded by the public. As the software developed in this present work is intended to be just a proof-of-concept prototype, the present extension is not released to the public but can be installed as follows:

1. Navigate to chrome://extensions in the Google Chrome browser
2. Check the box next to Developer Mode
3. Click "Load Unpacked Extension" and select the directory that contains the code of the extension to be installed

[4] https://developer.chrome.com/extensions

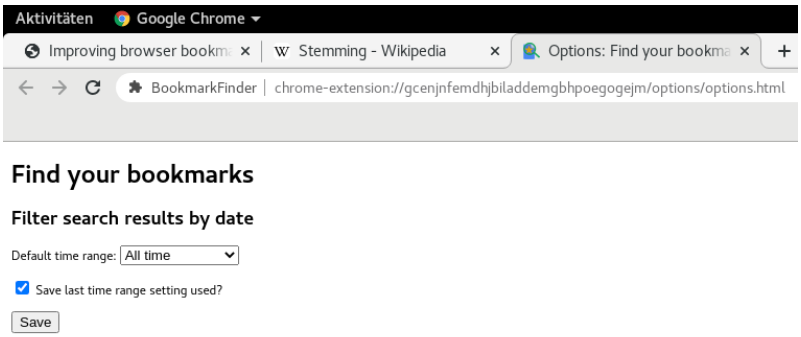

Figure 3.3: The options page provided by the extension in order to allow for customisation by the user.

Extension Options

An options page allows the user to customise an extension. Such an options page can provide various form fields or other HTML markup. The corresponding user input has to be persisted as explained earlier. According to the Chrome developer guide[5], an extension's options page can be reached by right-clicking the extension icon and then selecting options from the context menu or via the extension management page[6]. For convenience, the extension's popup can also link directly to the options page by calling `chrome.runtime.openOptionsPage()`. In this particular implementation such a page is used to allow for setting preferences concerning the date filter function, as can be seen in Figure 3.3.

Lunr.js

The lightweight full-text search framework Lunr was used to build the search index in this present work. The underlying concepts of Lunr have already been described in Section 2.1.8.

Lunr also supports so-called pipelines. A pipeline maintains "an ordered list of functions to be applied to all tokens in documents entering the search index and queries being ran against the index"[7]. Such pipeline functions can be added to

[5] https://developer.chrome.com/extensions/options
[6] chrome://extensions
[7] https://lunrjs.com/docs/lunr.Pipeline.html

improve an application that employs Lunr in a flexible manner. For example, words with the same meaning but different spelling like the British grey and American gray can be normalised using a specific pipeline function, in order to allow for retrieving results that match one of both variants when searching for one of them.

Another example of such a function is the stop word filter[8] that Lunr applies by default. This pipeline function filters common words that usually have no relevance in order to prevent them from being indexed. The default stop word list provides words of the English language only, but can be extended or customised for other languages as well.

Lunr Mutable

As a consequence of reducing index size and thus memory usage, Lunr does not allow for updating an existing index, that is without rebuilding the index completely whenever a change has to be made. Therefore, the default implementation is not well suited for our use case, as the index would have to be rebuilt every time a new bookmark is added to a user's bookmark collection. An index with such a behaviour is called unmutable. Conveniently, there exists a library called Lunr Mutable Indexes[9] that adds the required flexibility to Lunr. Although the index size tends to become larger in contrast to the default Lunr index, this is a reasonable trade-off.

Date Filter

Search results can be narrowed down further by filtering them by date. A simple drop-down field as shown in Figure 3.4 functions as an interface to let the user select a timespan for which search results should be displayed. When a date range other than "All time" is selected, the search results list only those bookmarks that have been saved or updated within the selected range, where the most recent saving or updating action is used for comparison. The extension's options page, as explained in Section 3.2.1, provides the possibility to choose a default date range which is then loaded by the popup on every start.

[8] https://github.com/olivernn/lunr.js/blob/master/lunr.js#L1194
[9] https://github.com/hoelzro/lunr-mutable-indexes

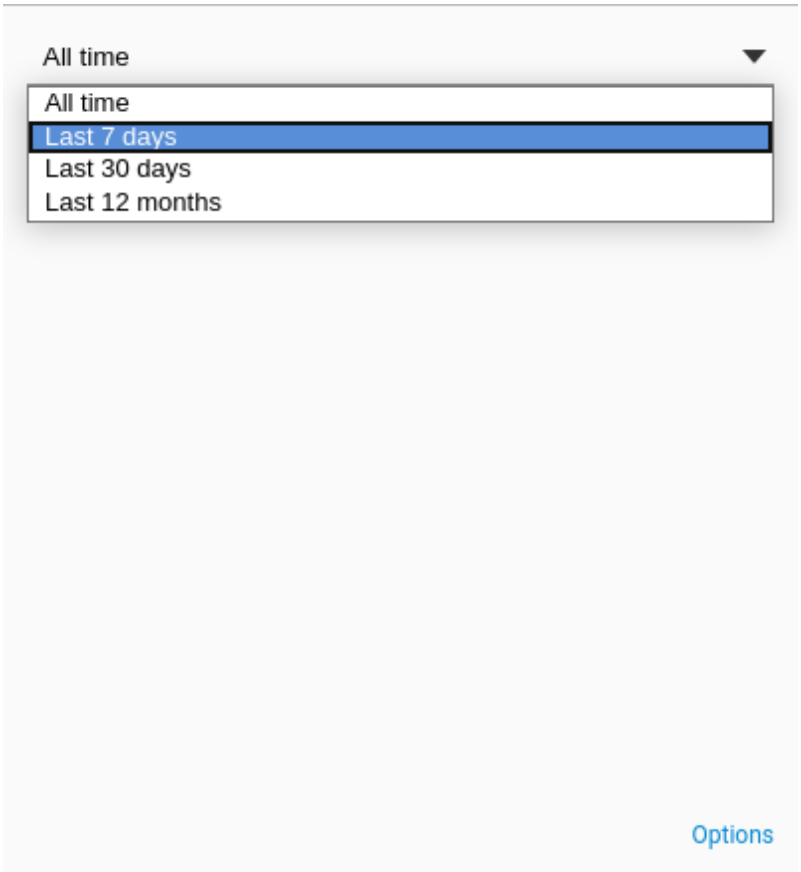

Figure 3.4: A simple drop-down list provides an option to filter search results by date. Only those of the retrieved bookmarks that have been saved or updated within the selected timespan will be displayed.

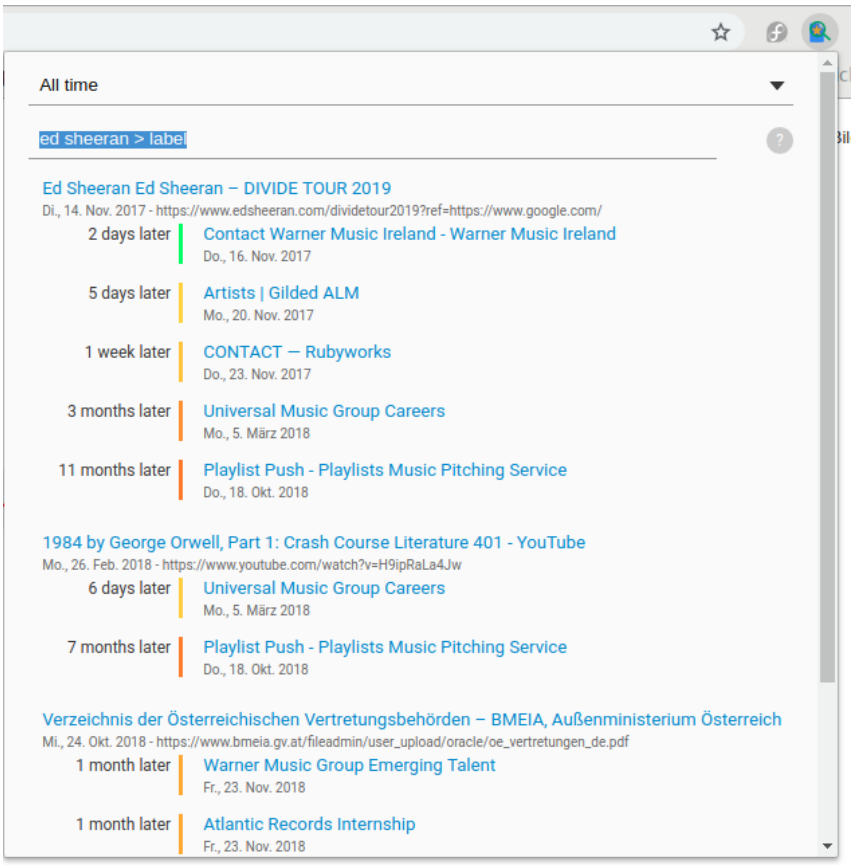

Figure 3.5: Showing results for bookmarks that match the term “label” and have been saved shortly after bookmarks that match either (or both) of the terms “ed” or “sheeran”.

Temporal Search

The temporal search functionality allows for search queries to include temporal context between bookmarks, which may prove beneficial to revisitation as explained earlier in Section 1.2.1. In the present implementation, the user interface is designed to be as simple as possible, therefore a temporal search can be initiated the same way as a basic query. However, in order to express temporal context in a query, the greater-than or less-than sign have to be used, where *term1* > *term2* can be interpreted as *results for term2 that have been saved (or updated)* ***shortly after*** *term1*. Consistently, a query searching for *results for term2 that have been saved (or*

updated) ***shortly before*** *term1* can be expressed as *term1 < term2*.

In our implementation of this functionality, the first step after the query has been initiated and parsing the user's input, is to start two separate queries, one for each side of the greater-than or less-than sign, respectively. The results of each query are then stored in the model, which in turn notifies the view using the observer pattern. On notification, the view renders the results as can be seen in Figure 3.5. The results of the second query are filtered, so that only results that were stored or updated after or before, respectively, each result from the first query are being shown. Furthermore, the results of the second query are sorted by date instead of being sorted by the score of a bookmark for the given query in order to emphasise the temporal context of the results. Additionally, each of the results of the second query is labelled using a traffic light colour scheme, where green means very shortly after or before, respectively, and orange indicates a longer timespan between saving two bookmarks. The calculation of each result's colour is shown in Listing 3.10, employing the HSL colour model since the saturation and brightness should be the same for all results, but the hue needs to be changed according to the respective timespan. As we hypothesise that the results which are closest with respect to their saving or updating date are the ones the user aims for, the colouring is calculated using a logarithmic scale in order to highlight these closest timestamps.

Listing 3.10: Colouring of temporal search results using a logarithmic scale on the HSL colour model.

```
for(var i in tempSearchResultBlocks) {
  var val = tempSearchResultBlocks[i].dataset
  if(val) {
    val = val.tempsearchTimediff

    //logarithmically scale between 0 and 1
    var norm = normalize(val, maxDiff, minDiff)

    //use HSL colour model, since we just want to change the hue
    var hslGreen = 140 //degree
    var hslOrange = 30
    var diff = hslGreen - hslOrange

    //calculate amount to go from green into the orange direction
    var normDiff = norm * diff
    var h = Math.floor(hslGreen - normDiff).toString()
    tempSearchResultBlocks[i].setAttribute("style",
        "border-color: hsl(" + h + ", 100%, 50%);")
  }
```

Figure 3.6: A gradient given in the HSL colour model with a hue of 20 and full saturation. The brightness runs linearly from 13 to 161 from left to right. A specific brightness value where brown ends and orange begins, or vice versa, can hardly be defined.

}

Search by Colour

The present work tries to allow for querying bookmarked web pages by colour. The colour codes are extracted by the content extraction server as described in Section 3.2.2 and sent to the browser extension which then processes the colour codes. The decision was made to extract colour terms in natural language from a given web page's colour scheme, in order to be able to add these terms to the index. This allows for querying in natural language and calculating a score based on term frequency instead of having the user have to input colours using some kind of colour picker and searching for similar colours, resulting in a simpler and more natural search experience for the user. However, as already stated in Section 1.2.1, this does not pose an easy task at all. For example, Figure 3.6 illustrates the difficulty of precisely assigning natural language terms to colour codes expressed in the HSL colour model, by showing that it is hardly possible to exactly separate the colours orange and brown by defining a certain point on the brightness scale.

A simple approach to this problem would be to let natural language terms overlap, that is assigning multiple terms like, for example, "brown, orange, red" to one colour that lies in an area of uncertainty. However, this approach requires to pay caution as overlapping areas that overlap too much are not optimal with respect to precision, since many of these colour terms would have high inverse document frequency as a result. On the other hand, naming one colour code "orange" while some user interprets the colour to be "red", this user will not be able to find the demanded web page, thus leading to low recall.

Another example of divergent perception of colour is grey. Grey usually is a colour of any hue or brightness (except no brightness and full brightness, which is black and white, respectively) but with no saturation. Or expressed in the RGB colour model, a colour where each R, G and B channel has equal values. However, there are people who will perceive a slightly saturated colour as grey as well. This effect is even stronger especially for rather dark as well as rather light colours. In general, a certain value of saturation that leads to the hue not being able to be perceived can not be defined, though. Again, the assignment of the tag "grey" should be kept to a minimum so that only those colours are defined to be "grey" that have a realistic chance of being perceived as such.

Therefore, a trade-off has to be made between precision and recall. Listings 3.12 and 3.11 states how this trade-off was implemented in the present work. The stated values are intended to serve as a starting point and are subject to further evaluation. It is also worth noting that colour perception is not only an issue of cognition and perception on the user's side, but instead also depends heavily on the hardware configuration, like the computer monitor and its calibration, a colour is viewed with.

Another issue that is being adressed in Listing 3.11 is different terms even in the very theoretical case that two distinct users could perceive the same colour. For example, one user might name some colours red, blue and green, but another user differentiates between barn red, carmine, fire brick and scarlet.

Listing 3.11: A starting point for defining hue value ranges. It can be easily seen that some ranges overlap.

```
function ColorManager() {
  this._hues = [
    {'color': ['red'],
     'range': [345, 360]
    },
    {'color': ['red'],
     'range': [0, 10]
    },
    {'color': ['orange', 'gold', 'brown', 'beige', 'ivory'],
     'range': [0, 40]
    },
    {'color': ['yellow', 'gold', 'beige', 'ivory'],
     'range': [35, 70]
    },
    {'color': ['green', 'lime'],
```

```
      'range': [65, 160]
    },
    {'color': ['cyan', 'aqua', 'turquois', 'mint', 'green'],
     'range': [140, 200]
    },
    {'color': ['blue', 'navy', 'indigo', 'teal'],
     'range': [175, 240]
    },
    {'color': ['purple', 'plum', 'indigo', 'pink'],
     'range': [230, 290]
    },
    {'color': ['pink', 'magenta', 'rose', 'purple'],
     'range': [270, 355]
    }
  ]
}
```

Listing 3.12: A function to match HSL colour codes to natural language terms, using margins to adress divergent colour perception. For example, black is defined not only when there is absolutely no brightness, but instead also for brightness values up to 3 out of 100, for a user usually can not perceive any hue for such values.

```
matchHSLtoString: function (hsl) {
  var h = hsl[0]
  var s = hsl[1]
  var l = hsl[2]

  var colorStrings = []

       if(l <  3) colorStrings.push('black')
  else if(l > 95) colorStrings.push('white')
  else if(s <  1) colorStrings.push('grey', 'gray')
  else {
      colorStrings.push((l < 50) ? 'dark' : 'light')

      /* rather dark or light colours oftentimes do
      ** not allow for perceiving a specific hue.
      */
      if(s < 30 && (l < 20 || l > 80))
          colorStrings.push('grey', 'gray')

      /* however, the hue is saved in any case, so that
      ** both 'kinds of perception' are being adressed
      */
      for(i in this._hues){
```

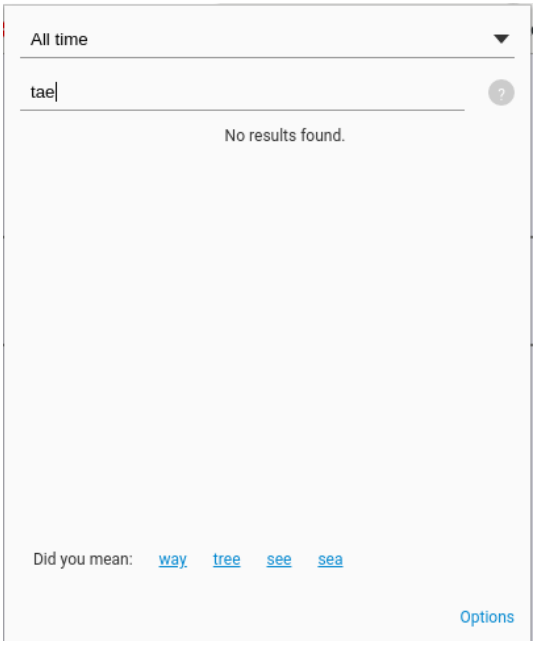

Figure 3.7: On user input, the extension tries to suggest queries by expanding and fuzzy matching the user's term.

```
            let hue = this._hues[i]
            let hueStart = hue.range[0], hueEnd = hue.range[1]
            if(h >= hueStart && h <= hueEnd){
              hue.color.forEach(function(color) {
                colorStrings.push(color)
              })
            }
        }
    }

    return colorStrings
}
```

Spelling Correction

A spelling correction or "Did you mean" functionality is well known from search engines, where user input gets checked and, for example, typos can be recognised.

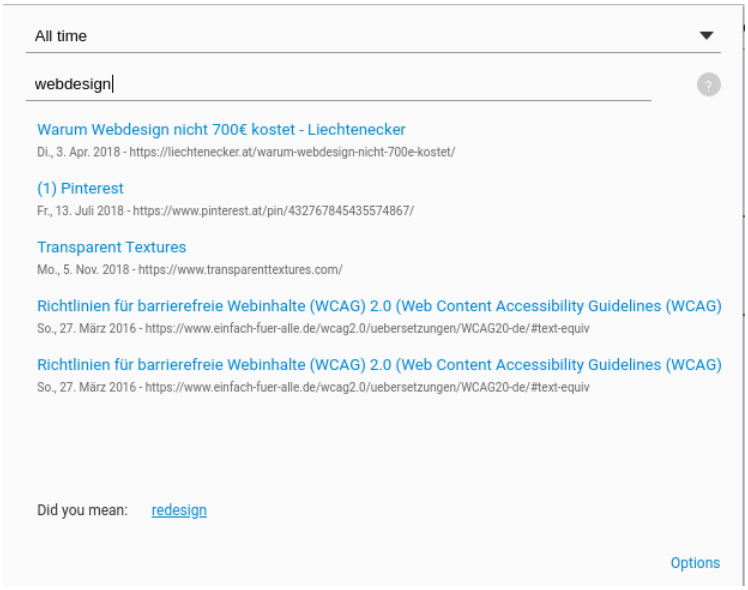

Figure 3.8: On user input, the extension tries to suggest queries by expanding and fuzzy matching the user's term.

But this function is not only limited to typos, instead search terms with similar meanings can also be suggested to the user. In our rather simple implementation, a fuzzy match query is being started upon every user input. This query searches for results with an edit distance of two to the user's input, as shown in Listing 3.13 taken from our implementation's controller. An edit distance of two means, that terms that differ from the search term by up to two characters still match the search term. For example, with an edit distance of two, the term "java" would match the term "jvm", since they only differ by two characters, that is "jvm" misses one "a" and has the other "a" replaced by an "m".

The metadata field is then extracted from the fuzzy matched search results' matchData field. This metadata field contains the token which matched the query. The original query token is then filtered out of the fuzzy matched tokens and four of the resulting suggestions are then returned. After saving these four suggestions in the model, the view presents these suggestions to the user as shown in Figures 3.7 and 3.8.

Listing 3.13: Search for terms with an edit distance of two. The function extracts the metadata field from each search result's matchData field and filters out the token of the original query, then returns four suggestions.

```
function didyoumeanSearch(searchTerm, idx) {
  var didyoumean = []

  try {
    var results = idx.search(searchTerm + "~2")
  } catch(e) {
    console.warn("CONTROLLER> invalid Query?! (" + query + ")")
    return
  }

  if (results.length) {
      didyoumean = didyoumean.concat(
          results.map(function(v, i, a) { // extract
             return Object.keys(v.matchData.metadata);
          }).reduce(function(a, b) { // flatten
             return a.concat(b);
          }).filter(function(v, i, a) { // uniq
             return a.indexOf(v) === i;
          }).filter(function(value){ return value !== searchTerm})
          .slice(0, 4)
      )
  }

  return didyoumean
}
```

Autocomplete Queries with the Help of a Web Worker

An autocomplete function for the query interface can enhance the search experience for the user by suggesting search terms. For example, by providing terms based on expanding or fuzzy matching the current term entered by the user. In order to implement such a function, in this present work, a web worker as described in Section 3.1.4 has been made use of. Finding search terms that can be suggested to the user, that is, querying the index with additional term expanding and fuzzy matching, can be a time-consuming task. This is due to the fact that the autocomplete function should suggest terms as the user types, as shown in Figure 3.9. This means, on every keyboard event a query is started to search for autocompleted suggestions. For

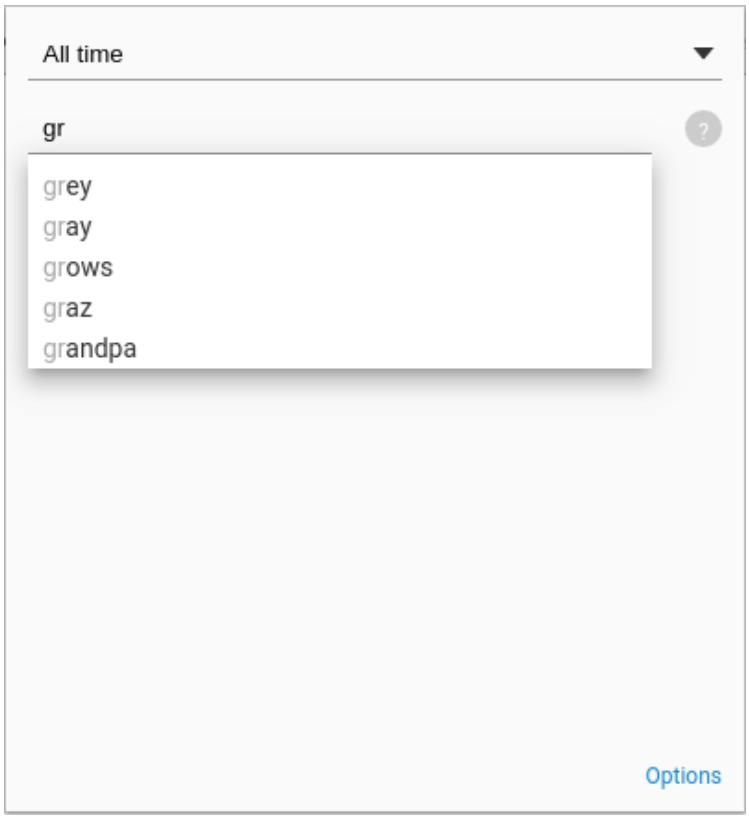

Figure 3.9: On user input, the extension tries to suggest queries by expanding and fuzzy matching the user's term.

example, when a user searches for "flower", the autocomplete function is triggered six times, querying for "f", then for "fl", "flo" and so on. Since this is not necessarily helpful in the case when the user quickly enters a word and only leads to even more overhead and therefore more blocking potential, a timeout threshold of 300ms has been defined as shown in Listing 3.14. If the query gets updated again within the timeout, the clearTimeout() function removes the worker's job from the event loop. Then a new instruction with the now updated query is added. In the above mentioned example, the autocomplete function is invoked only once for the term "flower", if the user quickly types the whole term at once.

Listing 3.14: The controller's udpateQuery function: On every query update, instruct the worker with searching for suggestions after a 300ms timeout. If the query gets updated again within the timeout, the clearTimeout() function removes the worker's job from the event loop. Then a new instruction with the now updated query is added.

```
updateQuery : function (query, triggerAC) {
  var _this = this
  _this._model.setQuery(query)

  clearTimeout(_this._acDelay)
  if(triggerAC) {
    _this._acDelay = setTimeout( function() {
      if(query !== "") {
        _this._acWorker.postMessage({"query": query})
      }
    }, 300)
  }
}
```

However, this timeout technique solves the issue only to a certain degree and is still insufficient for large indices, hence longer query times. Therefore, without the help of a web worker the GUI still would potentially block due to the single-threaded architecture of JavaScript described in Section 3.1.4.

The web worker is loaded when the controller of the popup is initialised and then communicates with the controller. The index needs to be sent to the web worker via its postMessage interface since the `chrome` API object is not available in the web worker and therefore the web worker can not load the index from the chrome local storage itself. The sequence diagram shown in Figure 3.10 illustrates how the MVC architecture and the web worker are tied together. For reasons of clearness, the diagram only shows the case when the index has already been created. In the opposite case, additionally the background script would have been needed to be invoked to create and return the index before starting the web worker.

3.2.2 The Content Extraction Server

The fundamental architecture of Node.js has already been described briefly in Section 2.1.1. The following sections describe how various Node.js modules are tied together to extract textual content as well as the primary colours used by a website from a given URL.

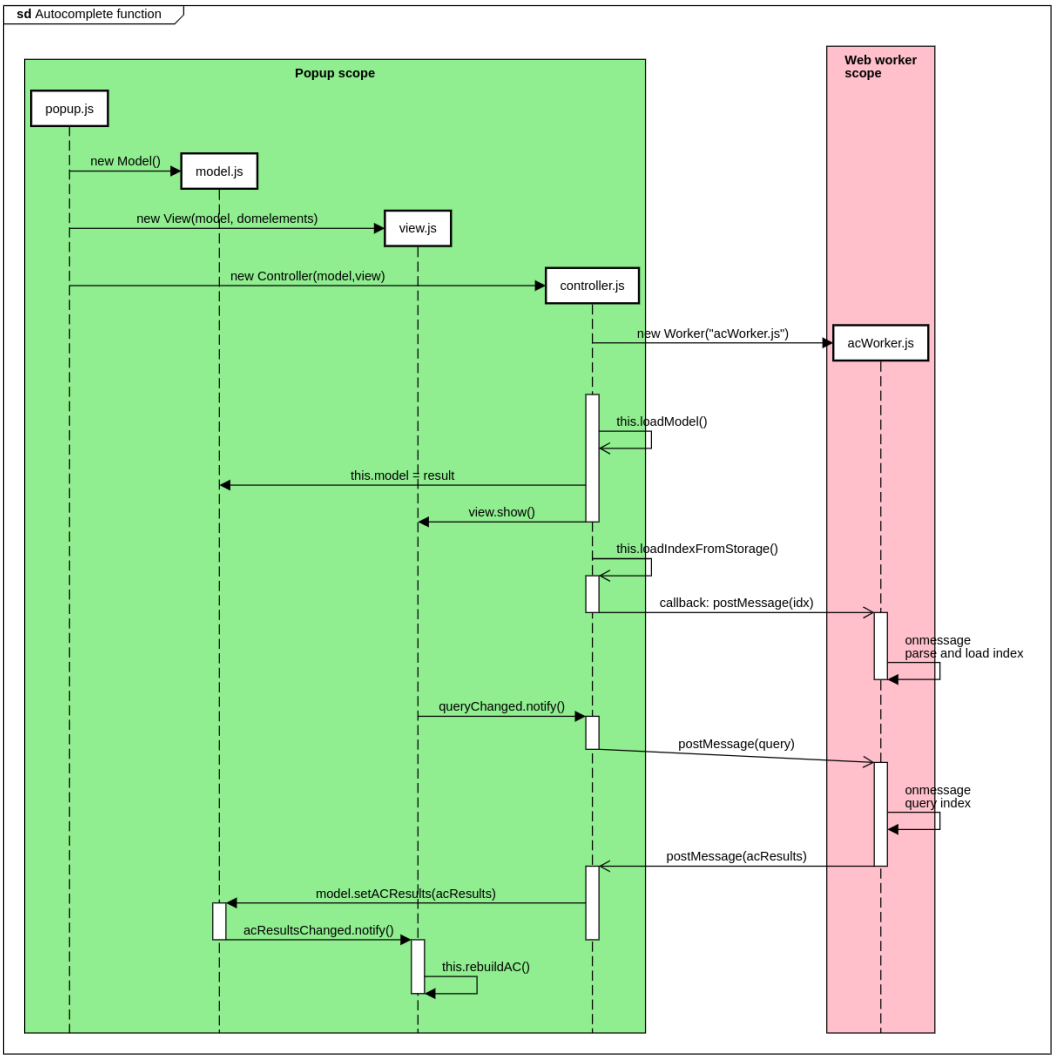

Figure 3.10: The sequence diagram illustrates the architecture of the autocomplete/autosuggest functionality. The web worker provides the controller with query results in a non-blocking manner. Note that solid arow heads represent synchronous, open arrow heads represent asynchronous method calls.

The HTTP Module

The HTTP module is compiled directly into the Node.js binary and as such is part of the default Node.js runtime. It can be used to easily create a simple HTTP webserver as shown in Listing 3.15. The created server object listens on the specified port and manages all connections. Incoming requests are handled by interacting with the Tika module to parse the contents of the URL-encoded URL transmitted using an "url" GET parameter. The content extracted by the `parseWithTika(url, date, callback)` function is then sent back to the requesting client by using a callback function.

Listing 3.15: An HTTP server object created by the HTTP Node.js module manages client connections on port 8080 and interacts with the content extraction function. The content extraction function parseWithTika() then responds to the client with the extracted content by using a callback function.

```
http.createServer(function (req, res) {
  var q = url_mod.parse(req.url, true).query;
  var source = q.url,
      date   = q.date
  parseWithTika(source, date, function(text){
    try{
      res.end(text)
    }
    catch(err){ console.log(err.message) }
  })

}).listen(8080)
```

The Node-tika, Webshot and Colorthief Modules

Node-tika, the Node.js port of Apache Tika, has already been presented briefly in Section 2.1.2. In the previous section, the calling of the function `parseWithTika(url, date, call` has been shown. An essential part of this function is to execute Tika's `extract(uri, options,` function as shown in Listing 3.16. Basically, the extraction function just has to be provided with a URI. Before the actual content extraction, the Tika module then manages locating, retrieving and parsing of the resource. Tika is able to automatically recognise a variety of supported filetypes, ranging from simple HTML web pages to PDF files or even image files, which can be processed using optical character recognition (OCR) software.

Additionally to Tika's content extraction, the anonymous callback function provided to the extract() function makes use of two other Node.js modules, namely color thief node[10] and webshot[11], which work together to render a screenshot of a given resource and then extract the most dominant colours out of the image.

Listing 3.16: Upon finishing the processing of a provided URI by Tika, further processing and colour extraction is being done by making use of the webshot and color thief modules and various nested callback functions.

```
function parseWithTika(url, date, callback){
  tika.extract(url, tikaOptions, function(err, text, meta) {
      var content = }
      if(typeof text !== 'undefined'){
        //using regex to clean all blank lines and tabs
        text = text.replace(/^\s*[\r\n]/gm, "")
        text = text.replace(/\t/g, ' ');
        var textlines = text.split("\n")

        for(line in textlines)  //trim lines
          content = content.concat(textlines[line].trim().concat(" "))
      }

      var metaString = undefined
      if(typeof meta !== 'undefined' && meta) {
        metaString = }
        if(meta.keywords)
          metaString = metaString.concat(meta.keywords[0])
        ...
      }
      ...
      webshot(url, img, options, function(err) {
        try{
            var colorThief = new ColorThief()
            var dominant = colorThief.getColor(img)
            var palette  = colorThief.getPalette(img, 3)

            //merge dominant into palette colours
            palette.push(dominant)

            try{ //delete webshot
              fs.unlink(img, function(err) {...})
            } catch(err) { ... }
```

[10] https://www.npmjs.com/package/color-thief-node
[11] https://www.npmjs.com/package/node-webshot

```
        } catch(e) {
          ...
        } finally {
            var parsed = {
                "content": content,
                "meta": metaString,
                "colors":  palette //in RGB
            }
            //use callback function on extracted content
            callback(JSON.stringify(parsed))
        }
      })
  }
}
```

4 Evaluation

The main purpose of this qualitative evaluation was to determine if our browser extension proves to be beneficial to a user's revisitation experience. However, since there are already tools that are similar to our work, a special focus was placed on investigating the functionality that is specific to this present work, that is, mainly the colour and temporal search functions. Of course the participants of the evaluation have been asked to evaluate the traditional search functions like the date filter as well, though, in order to be able to compare those functions to the more advanced search functionalities.

In order to be able to provide for a better understanding of the results, hereinafter the term "traditional search functionality" refers to date filter, autocomplete and "did you mean" functions, not including trivial text search, whereas the term "advanced search functionality" refers to the temporal search and colour search functions.

This goal led to the following narrowed down research questions: a) Does the usage of the more advanced search functions improve the findability of bookmarks that have content that is similar to lots of other results, b) Does the usage of the more advanced search functions help to increase the subjective certainty of having found the right bookmark, c) Does the usage of the more advanced search functions help to reduce the number of queries needed to find the desired bookmark, d) Do users memorise dominant colours of a web page, e) Do users think in temporal episodes, f) Does a bookmark search browser extension have potential to increase the overall extent of the usage of the bookmark feature, that is, make a browser's bookmark function more practical, and g) What type of search interface proves to be the subjectively most helpful.

4.1 Method

4.1.1 Participants

Ten test users, of whom 50% were female, completed the qualitative evaluation. The participants, with a mean age of 28.5 years (standard deviation = 10.6, median = 24.0), had a mean of 16.5 years of experience with computers (SD = 7.4) at the time of completing the study. All participants stated to use the web on a regular basis, with a mean of 14.6 hours per week (SD = 8.0), and predominantly use the Google Chrome browser, either as the one and only browser they use or besides some other browser, with only one participant not using Google Chrome at all.

Out of the ten participants, three stated to have a bookmark collection of less than 30 bookmarks, the largest share, namely five users, stated to have a bookmark collection of more than 30 but less than 100 bookmarks and one user explained to have a rather large bookmark collection of more than 500 bookmarks. Only one user did not use bookmarks at all.

4.1.2 Preparations

Two bookmark collections containing 53 and 42 bookmarks, respectively, have been artificially generated and used for the evaluation. Both bookmark collections have been designed by using a persona for each collection. The goal of this method is to make the bookmark collection that has to be evaluated, though being created artificially, feel like it has been generated organically by some real person. Each of the two personas were given some background parameters like age, occupation and hobbies, as well as a bookmark collection corresponding to these attributes, to make the bookmark collections to be evaluated relatable and plausible to the participants. In addition, the bookmark collections have been designed to allow for creating use cases that can not be solved easily by using traditional full-text search, but instead demand search functionality implemented in this present work, like colour search or temporal search. However, it has been paid particular attention to make the use cases realistic and plausible, though.

Furthermore, the timestamps of the bookmarks have been manipulated in such a way as to make the temporal contexts between individual bookmarks feels as natural

as possible. This manipulation was possible by exporting the created bookmark collection as an XML file, adapting the timestamp tags within this file and then reimporting that file to the Google Chrome browser.

After the two bookmark collections have been created, a bookmark "catalogue" in the form of a PDF file was compiled out of each bookmark collection. The pages of the two catalogues have been sorted chronologically within each catalogue, starting with the oldest bookmark of the regarding bookmark collection. Each page of these two PDF files showed a screenshot of the corresponding bookmarked web page, the domain part of the URL and the (manipulated) time of the creation of the bookmark. Additionally, the timestamp was also displayed in a more human-readable format in the form of, for example, "19 months ago - Friday noon". An example of such a page of the bookmark catalogue is shown in Figure 4.1 The aim of reorganising and presenting the bookmark collections in this way, was to help the participants get a better overview of the rather large bookmark collections.

Google Chrome allows for setting up multiple user profiles, so it sufficed to create each persona only once in form of a browser profile, including the corresponding bookmark collection, and then swap between these profiles as needed. After setting up the profiles, the participants of the evaluation were presented with the browser already opened with the desired user profile and our browser extension already preinstalled. It is worth noting that the indexing was already completely done upfront, before the participants were asked to use the extension.

Personas

The two personas each consisted of a short description of their personality, occupation and hobbies or interests, as well as four use cases that were to be tested. In order to avoid problems understanding the personas and their tasks, the original persona descriptions were formulated in the participants first language, German. However, an English translation is given below:

Harald, 45, Manager

You are a 45-year old manager interested in sports and wine. Recently you are especially into American football, furthermore you have a particular interest in sports cars - could it be that you are in the middle of a midlife crisis?

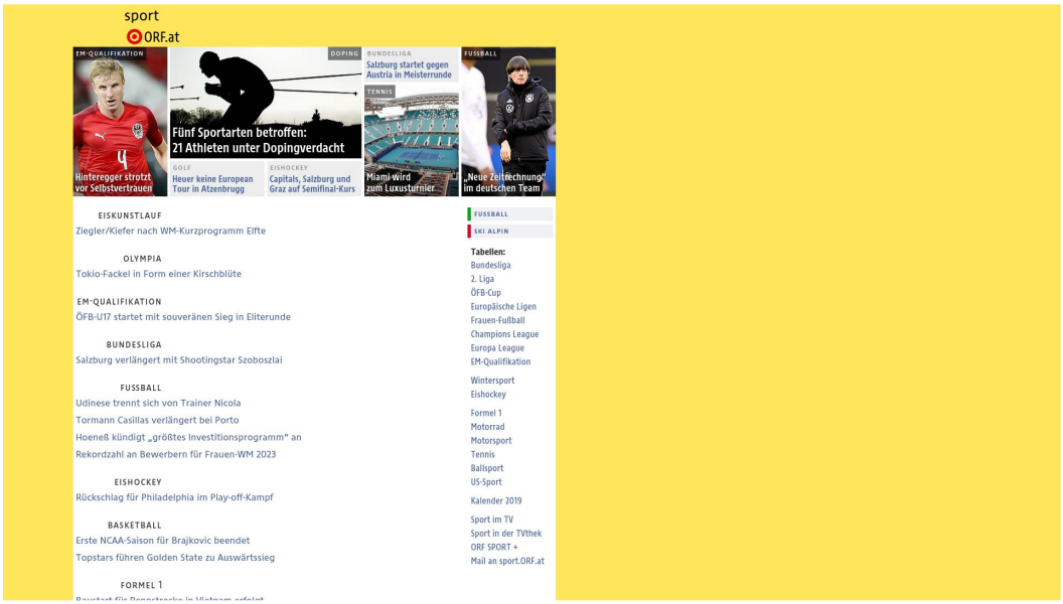

Figure 4.1: The bookmark collections used in the evaluation have been compiled into a PDF file each for a better overview. The figure shows one page of such a catalogue file. Each page shows one bookmark, including the domain part of the bookmark's URL, a timestamp, a more human-readable format of the timestamp and a screenshot of the bookmarked web page.

The corresponding use cases, translated into English, were as follows:

1. You want to order wine online, but you can not remember the name of the winery in the city of Horitschon.
2. Find the price of the vinyl record you have bookmarked last.
3. Search for the American football team located in Graz.
4. After you searched for used sports cars, you found a 1991 model Ferrari. Find the bookmark.

Sally, 21, University student

You are a 21-year old university student of English and American studies and play in a punk rock band. Naturally, you are interested in (live) music and always looking for chances for studying or doing internships abroad.

The corresponding use cases, translated into English, were as follows:

1. Find the record label of Ed Sheeran.
2. You have bookmarked the website of your favourite band, The Flatliners, shortly before going to a show of theirs in a music club. After the show, you also bookmarked the club's website. Find the website of the club.
3. Find your latest bookmark related to the topic "Uni Graz".
4. Search for information about EU student exchange programmes at your university.

4.1.3 Procedure

The participants were invited separately to carry out the following procedure. First of all, each participant was asked to fill out a short form in order to assess if they were familiar with using computers and the world wide web in general, as well as using bookmark functionality in particular. After that, each participant was shown the description of one randomly selected persona, as well as the assigned persona's "bookmark catalogue". The participants were told that they did not have to study the catalogue by heart, but instead should take their time to get a feeling for the bookmark collection and keep an eye on the timestamps and try to get a rough sense of the intervals and temporal contexts between the bookmarks. Before heading on to the use cases, the participants were allowed to try out the browser extension and explore its functionality for a few minutes.

After the participants got familiar with the browser extension's functions, they were asked to carry out the four tasks corresponding to their assigned persona. However, there was one single restriction, to that effect that each participant was asked to carry out some specific tasks only with traditional search functionality or simple full-text search, whereas the remaining tasks should, if possible, explicitly be approached by using more advanced functions, namely temporal or colour search. In order to compare the results depending on the used functionality, the tasks had been shuffled for each particpant. Furthermore, the method that should be used to carry out a specific task, that is, either using the traditional functions, also including trivial text search, or the more advanced functions, was chosen randomly. However, it was tried to aim for a distribution of approximately 50% between traditional and advanced functions. For the 40 tasks, resulting from 10 participants solving 4 tasks each, the actual share of tasks where the participant was encouraged to use advanced functionality was 22 tasks (55%). For the remaining 18 tasks, participants were asked to solve the task by using traditional methods or trivial full-text search. Additionally, where possible, it was tried to arrange the tasks in order of their difficulty, starting with the easier ones.

After completing each task, the participants were asked to rate their certainty about having found the demanded bookmark on a 7-point Likert scale.

Furthermore, upon completion of all four tasks, each participant was asked to answer a feedback questionnaire, consisting of questions concerning the respective user's overall search experience using the tool. The questionnaire contained open questions asking about things that struck the participant as particularly good or bad as well as a multiple choice question if the participant could imagine a function to generally have potential to be useful or not. A single choice question asked about which search function was actually the most helpful to the respective participant. Additionally, participants were asked to rate their overall satisfaction about getting to the bookmark they searched for, the speed of the plugin, how likely they are to use such a browser plugin in the future and how likely they are to pay for using such a plugin if it was sold in the browser store, each on a 7-point Likert scale.

4.2 Results

In order to approach the total of 40 tasks, participants used temporal search and colour search functions 9 times each. The date filter option was used 4 times whereas the autocomplete function was used only once. The "Did you mean" suggestions were never used by any participant. For the remaining 17 tasks, no specific function besides the simple text search was used.

Deriving from the participants' own ratings of their certainty about having found the desired bookmark, traditional functions were indicated to provide the highest possible certainty, having a mean rating of 3.0 on a 7-point Likert scale with values ranging from -3 to 3. Taking into account trivial full-text search as well, when looking at traditional functions including full-text search, a mean certainty rating of 2.44 was given. On the same scale, advanced functions scored a mean certainty rating of 2.0.

The number of queries needed to find the desired bookmark is another metric that was investigated. When using traditional functions, including trivial full-text search, an average of 1.42 queries were needed. In contrast, when using advanced functions an average of 1.61 queries were necessary to obtain a satisfying result. However, three times a participant could not obtain the desired result, or at least did not recognise the desired bookmark amongst a given set of search results, when asked to only use traditional search functions or trivial full-text search.

The previously stated values are given in more detail in Table 4.1, including standard deviations and each function's individual values. Additionally, Figure 4.2 shows a scatter plot of mean certainty and mean number of queries needed for each function.

When asked whether something was particularly good, participants stated that they especially liked the date filter (3 times), temporal search (2 times) and colour search (2 times) features.

Regarding the question if something was perceived as bad, two participants explained that the "help" tooltip explaining the usage of certain search paradigms, like temporal search, could be better, or at least be displayed in a more prominent manner, for example by being reminded of the available options or functions. Furthermore, one participant expressed the wish for a search history. Another participant presented the idea of introducing a keyboard shortcut for activating

Table 4.1: Comparison of search functions in terms of number of needed queries and user-rated certainty. It has to be considered, though, that no result could be obtained for 3 of the tasks where participants were asked to only use traditional search functions or full-text search. This fact was taken into account for computing certainty values by assigning the lowest possible certainty value of -3, but could obviously not be adressed for the number of queries needed.

	Certainty		Queries needed		Times used
	Mean	SD	Mean	SD	
Date filter	3.00	0.00	1.50	0.58	4
Autocomplete	3.00	-	1.00	-	1
“Did you mean”	-	-	-	-	0
Trivial full-text search	1.88	2.34	1.59	0.94	17
Temporal search	1.89	1.27	1.11	0.33	9
Colour search	2.11	2.03	2.11	1.27	9
Colour & Temporal search	2.00	1.64	1.61	1.04	18
Date filter, Autocomplete, “Did you mean” & trivial full-text search	2.44	2.10	1.42	0.86	22
Date filter, Autocomplete, “Did you mean” without full-text search	3.00	0.00	1.25	0.55	5

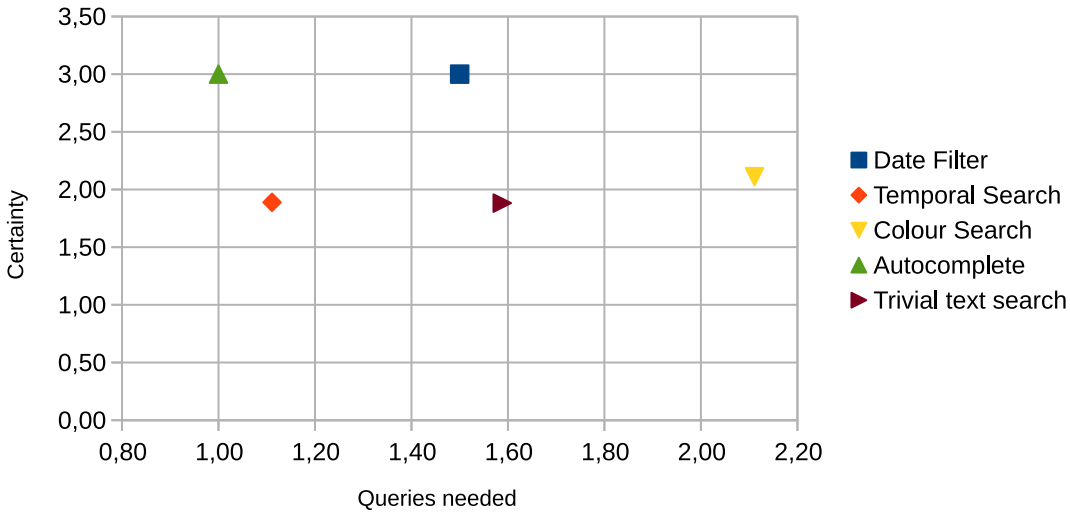

Figure 4.2: Scatter plot showing the user rated mean certainty and the mean number of queries needed for 40 tasks, by function. The goal is higher certainty and lower number of queries needed, thus, the functions farthest in the upper left corner are the best performing ones.

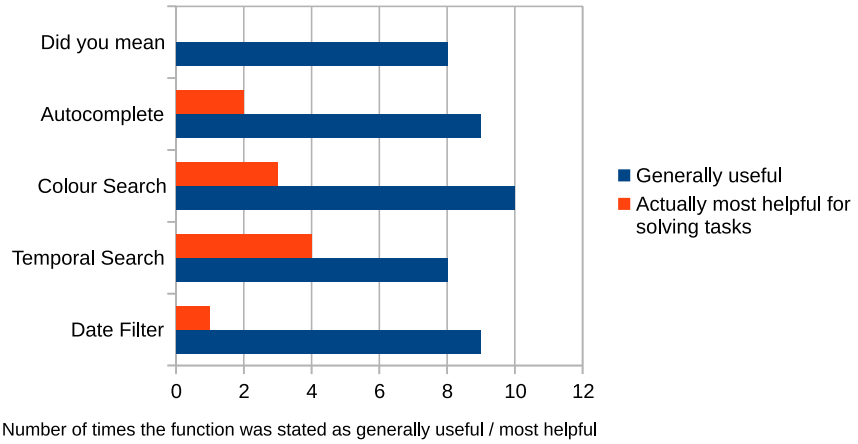

Figure 4.3: Bar chart showing the number of times a function was classified as potentially useful (multiple choice) and actually be the most helpful for completing four tasks (single choice), respectively, by ten participants.

the browser extension popup, which is actually possible to implement in Google Chrome[1].

After completing all four tasks, each participant was asked about the general usefulness of each function and which of the functions was the most helpful. The answers given to this questions are shown in Figure 4.3.

Finally, participants were asked to rate certain general attributes of the overall extension on a 7-point Likert scale with values ranging from -3 to 3. On average, participants rated the easiness of getting to the right bookmark with 2.3 (SD = 1.3), the overall speed of the browser extension with 2.7 (SD = 0.7) and the likeliness of using this or another similar tool in the future with 1.5 (SD = 1.3). When asked about the likeliness of paying for this or a similar browser extension, participants rated the likeliness with a mean of -0.3 (SD = 2.1). Furthermore, participants, on average, estimated the probability that such a browser extension would increase their usage of a browser's bookmark functionality to be 2.0 on the given Likert scale (SD = 0.8).

[1] https://developer.chrome.com/extensions/commands

4.3 Discussion

First of all, it should be mentioned that participants occasionally used more than one method, but only the method leading to the desired bookmark was considered for deriving results.

The reason that "Did you mean" suggestions were never used may be that these suggestions were being displayed beneath the search results and therefore were not brought to the participants' attention so well. Another reason may be that "Did you mean" suggestions are generated by fuzzy matching terms with a certain lexical distance, which, in combination with the Porter stemmer used in the present work, sometimes provides just meaningless arrangements of characters instead of actual words. Suggesting semantically more meaningful terms may present a better solution.

As can be seen in Figures 4.2 and 4.3, particpants' ratings of usefulness of a certain search function do not always correlate to certainty of a specific task when using the same function. This might be due to being used to traditional functions and therefore being biased towards new functions. However, another valid reason might be that advanced search functionality simply provides for a subjectively more pleasant search experience, which results in appearing more helpful to the user.

Looking at Figure 4.2, it looks like traditional search functions outperformed advanced search functions with respect to number of queries needed and a participant's certainty of having found the desired bookmark. However, it has to be considered, though, that no result could be obtained for 3 of the tasks where participants were asked to only use traditional search functions or full-text search. This fact was taken into account for computing certainty values by assigning the lowest possible certainty value of -3 on the Likert scale, but could obviously not be adressed for the number of queries needed. This leads to the suggestion that advanced search functions might perform better, or even find results where traditional functions can not provide any results at all, in certain special cases. Such cases might especially be use cases where some context, for example, temporal context, is needed to evaluate the results provided by the browser extension. Temporal search being rated by participants as the actually most helpful functionality, as can be seen in Figure 4.3, might reenforce this hypothesis further. On the other hand, some of the use cases were designed to require some context in order to find the bookmark in question.

That is, the concerning tasks were borderline cases, specifically designed to be hard to find without advanced functionality, though such use cases do exist in real everyday use.

4.3.1 Lessons learnt

Designing Use Cases

It sometimes is quite difficult to make up use cases that can be evaluated. First of all, the goal of this present work was to take advantage of a user's hypothetical ability to memorise temporal episodes or certain colours. Such knowledge is usually being held in a user's long-term memory. The evaluation methodology used in this present work's evaluation does not employ long-term memory, since this would lead to a significant increase of evaluation effort. Instead, test users are presented with an artificial bookmark collection shortly before the evaluation. This way, the bookmark information can only be held in and retrieved from short-term memory, which makes it hard to design use cases that yield reliable conclusions for the real world.

On the other hand, real users have some personal association with their saved bookmarks. Such a connection does not exist for an artificially generated bookmark collection which the test users were asked to memorise. Both problems have been approached by using a persona. Connecting an artificial bookmark collection with a persona, that is, generating a bookmark collection that feels like it has been created by a real user with certain features or interests, can help a test user to identify and empathise with the bookmarks, and thus, helping a test user to memorise a given bookmark collection.

However, it is certainly still easier for users to memorise their own bookmarks. The reason for needing to generate artificial bookmark collections is that performing an evaluation that uses a test user's own personal bookmark collection would have required to design use cases or tasks specific to that given bookmark collection. This method would have been impractical to evaluate as the results for such bookmark collection specific tasks would not have been comparable with each other.

Therefore, the difficulty in designing personas was to make up corresponding use cases that allow for being used in two scenarios, the first one being some kind

of baseline task that is intended to evaluate if a certain bookmark can be found with traditional search queries. The second scenario's purpose is to test if the performance can be improved further by incorporating advanced search queries like temporal search or colour search.

Concluding, it turned out to be quite a difficult task to find use cases that can be tested in both scenarios in order to have comparable results while ensuring that the use case is still relatable and plausible to the test users.

Formulating Research Questions precisely

To be able to do a good evaluation, it helps to formulate research questions as precisely as possible. That way, the evaluation can be designed appropriately, ultimately leading to much more interesting results.

Using Guttman Scales for Feedback Questionnaires

The evaluation quality could potentially benefit from exchanging questions in a Likert scale in favour of questions in a Guttman scale, where questions are arranged in such a way, that they escalate in specifity, in order to reduce the chances of distorted results. Sometimes this escalation is obscured by adding intermediate questions. The reason for this is that the respondents may tend to give biased answers, for example because they may tend to give answers biased towards the neutral or a positive answer.

4.4 Conclusion

In order to improve a browser's native bookmark functionality, solutions to various problems that still persist in existing state-of-the-art approaches have been elaborated in this present work. Following the investigation of typical issues, for example, requiring to give browser extensions access to the DOM or problems with revisiting automatically categorised bookmarks, a browser extension supporting full-text search was developed for the Google Chrome browser. Besides more traditional functions like simple date filters or an autocomplete function, the features of the

proposed browser extension also covers novel functions like colour search and temporal search. An evaluation with 10 participants was carried out, in order to investigate the performance and usability of the suggested approach. Participants' ratings (on average, particpants stated that the probability that such a browser extension would increase their usage of a browser's bookmark functionality to be 2.0 on a Likert scale with values ranging from -3 to 3) and positive comments, especially regarding temporal search and colour search, support the hypothesis that the approach suggested in this present work poses an improvement of browser bookmark revisitation regarding findability and usability, while still respecting the fact that a user's bookmarks are very personal data by storing a search index locally and not interfering with the browser's DOM. Furthermore, temporal search being rated the most helpful function by participants, revealed that allowing for temporal context within a search query has potential to provide meaningful results for special cases, where traditional approaches fail.

4.4.1 Limitations

Bookmark Collection Size and Time needed for Indexing

To get a better understanding for the possible size of a real bookmark collection, one real bookmark collection consisting of 967 bookmarks has been looked at in terms of the time needed to create an index and the amount of storage needed to persist the resulting index. Out of those 967 bookmarks that the extension has processed in 23.1 minutes, 956 have been indexed successfully, while the remaining bookmarks could not be processed either to the original website being offline, password-protection, et cetera, as described in the previous subsection. However, the time needed to process such a bookmark collection can only be a very rough estimate and is subject to fluctuations, obviously, since Tika's content extraction performance relies on things like web server response times.

The resulting index holding the contents of the 956 bookmarks had a size of 10,655,401 bytes. To put this number into context, Google Chrome's storage API provides a synced as well as a local storage area with a maximum total amount of 102,400 and 5,242,880 bytes of data that can be stored, respectively. Fortunately, the `chrome.storage.local` limit can be circumvented by setting the `unlimitedStorage` permission in the extension's manifest file, as described in

Section 3.2.1. However, by assuming a mean index size of $10{,}655{,}401/956 = 11{,}145.81 \approx 11{,}000$ bytes per bookmark, it can be seen that it is not possible to synchronise an index between multiple clients directly by using `chrome.storage.sync` for a bookmark collection consisting of more than $102{,}400/11{,}145.81 \approx 9$ bookmarks.

Inaccessible bookmarked Web Pages

It is the nature of the world wide web that web pages change or even go offline. Naturally, this also concerns bookmarked web pages, which means that sometimes it happens that a bookmarked web page becomes unreachable, either temporary or permanently. The bookmarked URL can still be found by the browser extension with the method proposed in this present work. However, it is not guaranteed that this URL is accessible. One approach to tackle this problem is proposed in Section 4.4.2.

Password-protected Web Pages

In contrast to web pages that become inaccessible over time, there are bookmarked web pages that can not be accessed by the browser extension, or the Node.js server used in this present work, respectively, in the first place, for example, due to being password-protected. It is important to understand that a distinction has to be made between these two scenarios when approaching this issues. One possible solution to dealing with web pages that are inaccessible in the first place is proposed in Section 4.4.2.

4.4.2 Future Work

As of now, the present work is clearly not in a state that is able to be released to the public. Before a public release, above all, there has work to be put into security, load balancing, et cetera of the backend server, currently implemented as a Node.js server. Alternatively, we suggest to find a way to incorporate the backend server into the browser plugin directly. It has to be evaluated if this would constitute a realisable approach.

In the following some further improvement suggestions are outlined.

Override Pages and Injections

Google Chrome extensions offer a range of features to change the default behaviour of or interact with the browser. One such feature is to override certain pages with a custom HTML page[2], including style sheets and scripts. A good use case for the present work could be to override the bookmarks web page with a custom page that provides, for example, a search interface.

Using so-called content script injections[3] poses another way an extension can interact with the browser. Using this feature, a script can be triggered whenever, for example, a Google search is done by the user. Said script could then display relevant results from a user's bookmark collection in addition to the results that are found by Google, to provide for personal results where applicable even though the user did not explicitly use the browser extension.

Attaching Thumbnails to Search Results

Attaching a thumbnail, that is, a low resolution preview image of a web page, to a search result and displaying it, for example, when hovering the mouse over the result could potentially help to increase certainty about whether a given result contains the desired information. However, it should be noted that storing thumbnails may use relatively large amounts of resources. This is particularly the case for rather large bookmark collections obviously. As this issue tends to depend on subjective preferences of a given user, this could also be implemented as an optional opt-in feature.

Colour Search User Interface

In order to avoid problems regarding varying perception of colour and subsequent ambiguous colour term assignments as mentioned in Section 1.2.1, the text-based approach used in the present work may be replaced in favour of a "colour picker" UI

[2] https://developer.chrome.com/extensions/user_interface#additional_features
[3] https://developer.chrome.com/extensions/content_scripts#declaratively

as shown in Figure 4.4. Upon a user choosing a colour to search for using the colour picker, one possible approach may be to let the browser extension search for the "nearest neighbours" of the queried colour. It has to be determined which colour model and other settings should be used to determine these nearest neighbours. It also has to be investigated if this approach would pose a practical solution in terms of usability. However, the approach would allow for obtaining results within a continuous scale, without being restricted to search only within given colour terms. It has to be evaluated if this method results in a better performance for colour search.

Typing colour terms into the search query text box just as regular search terms is an obvious advantage of the approach used in the present work, in terms of usability. However, it also has disadvantages when there is a need to differentiate between colour terms referring to a perceived colour of the website and colour terms that actually are textual content of a web page. For example, when a user searches for the company "Red Bull", web pages that visually appear to be "red" are not necessarily wanted. Fortunately, having Lunr.js store colour terms in a separate field of the index allows for solving this problem, for example, by requiring a prefix like "c:" in order to start searching for colour. Lunr.js then allows for filtering the fields to search in. This way, by querying for example "c:red", after parsing the prefix, searching for results can be narrowed down to just search within the index's colour field.

Caching bookmarked Web Pages

As described previously in Section 4.4.1, it can happen that a bookmarked web page becomes inaccessible. Caching the contents of a web page upon indexing could possibly present one valid way to deal with this issue. In view of large bookmark collections and available disk space, it has to be assessed though, to what extent web pages should be cached. For example, a distinction between plain text, HTML and further assets like styles, scripts and images would be reasonable. However, caching only parts of a website typically is a trade-off between resource consumption and user-friendliness. As said earlier, besides the question as to how much disk space is available in the first place, the extent to which a computer's disk space should be used depends heavily on the preferences of a given user. Providing different caching levels on the extension's settings page could present a reasonable approach.

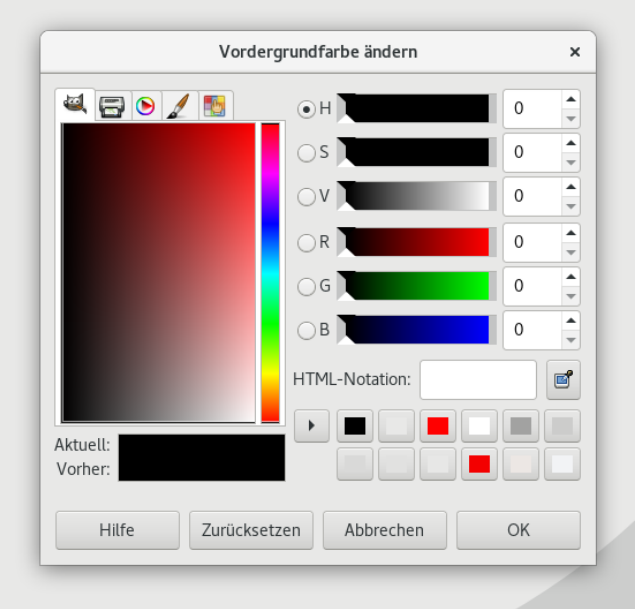

Figure 4.4: A screenshot of the GIMP 2.8.22 colour picker interface. This is a typical colour picker, comprising sliders, buttons, text fields and direct manipulation. The shown UI supports various colour models and representations.

Adding to Index manually

As described in Section 4.4.1, a limitation of the current method is that content can not be extracted of web pages that are not publicly accessible, for example, because they are password-protected, due to the browser extension not extracting contents from the DOM, but rather requesting the bookmarked URL autonomously. The reasons for the decision in favour of this behaviour have been explained in detail in Section 2.2.

However, providing the possibility to manually index (password-protected) web pages could pose a reasonable approach. With respect to data protection issues, such an opt-in solution should be preferred over opt-out alternatives. One possible approach to provide such an option is to keep record of the bookmarked web pages that already have been indexed and those that failed. The browser extension then, whenever the user visits some web page, has to check if the given web page is in the set of bookmarked URLs that failed indexing. If so, the user can be informed about their options regarding allowing the browser extension to add the currently viewed contents to the index.

Multilingual Bookmark Collections

With respect to multilingual use, the current work leaves room to improve. As described in Section 2.1.6, exchanging the currently used Porter stemmer in favour of an n-gram stemming algorithm could prove beneficial to relevance of retrieved documents. Incorporating language detection combined with language specific stemming algorithms poses another potential solution to said question. It has to be evaluated which approach proves to be more practicable.

While aforementioned suggestions regarding multilingual bookmark collections try to tackle general performance issues for situations where the user knows which language a certain query should be formulated in, there are situations where the choice of language is not that clear. For example, a user's bookmark collection consists of German as well as English websites. An index built with an n-gram stemming algorithm could potentially deliver pleasing results if queries are expressed in the "right" language. But the user probably is not sure what language the bookmarked website was in and therefore has a chance of missing results because of formulating the query in the "wrong" language. As a result, the user could try to formulate

the search query in both regularly used languages. Alternatively, so-called cross-language information retrieval (CLIR) holds potential to pose a more user-friendly search experience. The main idea of CLIR is to be able to retrieve results in a certain language by formulating queries in another language. One way to tackle this issue is a dictionary-based approach, as explored, for example, by (Qin et al. 2006) for the English–Chinese case.

Long-term Studies

Furthermore, in contrast to the evaluation methodology used in the present work, long-term studies incorporating test users' long-term memory promise to provide a better understanding of how users memorise bookmarks. Extending the method to evaluating users' own bookmarks holds potential to give an even better image of how users' tend to memorise bookmarks and formulate queries for such a long-term memory study. The proposed procedure is to have study participants install the browser extension on their own laptops, with the extension logging statistics about revisitation as well as adding a user interface that asks the participants questions about their experience and satisfaction with the extension. In order to avoid investing effort upfront into the security, scalability and availability of the currently used Node.js server running Tika, it should be evaluated how the content extraction part can be implemented directly in the browser plugin.

Appendix

Date: Time: User No.:

Background Questionnaire

Thank you for participating in our test. Please answer the following questions:

1. General Information

Sex: [ ] male [ ] female

Age: _____

Occupation: _____

2. Education

Educational Level Attained:

[ ] vocational training [ ] secondary school

[ ] university degree [ ]doctorate

If you are studying or have studied, please describe your main area of study:

_____

3. Use of Computers

How long have you been using a personal computer?

_____ years

How many hours per week do you use a computer?

_____ hours

Which kind of computer do you normally use?

[ ] Microsoft Windows [ ] Apple [ ] Linux [ ] Other _____

How many hours per week do you use the World Wide Web?

_____ hours

Which web browser do you usually use?

[ ] Internet Explorer [ ] Firefox [ ] Chrome

[ ] Safari [ ] Opera [ ] Other _____

How many bookmarks have you saved in your default browser?

[ ] < 30 [ ] < 100 [ ] < 500 [ ] > 500 [ ] I don‘t use bookmarks

Date: Time: User No.:

Feedback Questionnaire

Thank you for participating in our test. Please answer the following questions:

1. Did anything strike you as particularly good?

2. Did anything strike you as particularly bad?

3. Functionality

Did you find following search functionality useful?			Which one was the most helpful?
Date Filter	[ ] yes	[ ] no	[ ]
Temporal Search (X shortly after Y)	[ ] yes	[ ] no	[ ]
Search by color of website	[ ] yes	[ ] no	[ ]
Autocomplete search queries	[ ] yes	[ ] no	[ ]
Did you mean	[ ] yes	[ ] no	[ ]

3. Satisfaction

Please rate your satisfaction with these aspects of the plugin you have just finished working with, by circling the most appropriate number.

1. Getting to the right bookmark you searched for

Bad	-3	-2	-1	0	1	2	3	Good

2. Overall speed of the plugin (opening, search, autocomplete)

Slow	-3	-2	-1	0	1	2	3	Fast

3. How likely are you to use such a bookmark search plugin in the future?

Not likely	-3	-2	-1	0	1	2	3	Very likely

4. How likely are you to buy such a plugin if it was sold in the browser store?

Not likely	-3	-2	-1	0	1	2	3	Very likely

5. If you had this browser plugin available, would you use bookmarks more/more frequently?

Not likely	-3	-2	-1	0	1	2	3	Very likely

Bibliography

[1] David Abrams, Ron Baecker and Mark Chignell. 'Information Archiving with Bookmarks: Personal Web Space Construction and Organization'. In: (Apr. 2002). DOI: 10.1145/274644.274651 (cit. on p. 3).

[2] Simone Braun, Valentin Zacharias and Hans-Jörg Happel. 'Social Semantic Bookmarking'. In: *Practical Aspects of Knowledge Management*. Ed. by Takahira Yamaguchi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 62–73. ISBN: 978-3-540-89447-6 (cit. on p. 16).

[3] Andrey Chetverikov and Ivan Ivanchei. 'Seeing "the Dress" in the Right Light: Perceived Colors and Inferred Light Sources'. In: *Perception* 45.8 (2016). PMID: 27060181, pp. 910–930. DOI: 10.1177/0301006616643664. URL: https://doi.org/10.1177/0301006616643664 (cit. on p. 7).

[4] Andy Cockburn and McKenzie Bruce. 'What do web users do? An empirical analysis of web use'. In: *International Journal of Human-Computer Studies* 54.6 (2001), pp. 903–922. ISSN: 1071-5819. DOI: https://doi.org/10.1006/ijhc.2001.0459. URL: http://www.sciencedirect.com/science/article/pii/S1071581901904598 (cit. on p. 1).

[5] Marc Damashek. 'Gauging Similarity with n-Grams: Language-Independent Categorization of Text'. In: *Science* 267.5199 (1995), pp. 843–848. ISSN: 0036-8075. DOI: 10.1126/science.267.5199.843. eprint: https://science.sciencemag.org/content/267/5199/843.full.pdf. URL: https://science.sciencemag.org/content/267/5199/843 (cit. on p. 11).

[6] TV Do and RA Ruddle. 'MyWebSteps: Aiding Revisiting with a Visual Web History'. In: *Interacting with Computers* 29.4 (July 2017). © The Author 2017. Published by Oxford University Press on behalf of The British Computer Society. This is a pre-copyedited, author-produced PDF of an article accepted for publication in Interacting with Computers following peer review. The version of record Trien V. Do, Roy A. Ruddle; MyWebSteps: Aiding Revisiting with a

Visual Web History. Interact Comput 2017 1-22. doi: 10.1093/iwc/iww038 is available online at: https://doi.org/10.1093/iwc/iww038., pp. 530–551. URL: http://eprints.whiterose.ac.uk/110716/ (cit. on p. 3).

[7] Eelco Herder. 'Characterizations of User Web Revisit Behavior.' In: *LWA*. Citeseer. 2005, pp. 32–37 (cit. on p. 1).

[8] Pavlos Kokosis et al. 'HiBO: A System for Automatically Organizing Bookmarks'. In: *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries*. JCDL '05. Denver, CO, USA: Association for Computing Machinery, 2005, pp. 155–156. ISBN: 1581138768. DOI: 10.1145/1065385.1065419. URL: https://doi.org/10.1145/1065385.1065419 (cit. on p. 17).

[9] Jean-Éric Pelet and Panagiota Papadopoulou. 'Investigating the effect of color on memorization and trust in e-learning: The case of KMCMS.net (Knowledge Management and Content Management System)'. In: *Impact of E-Business Technologies on Public and Private Organizations: Industry Comparisons and Perspectives* (Jan. 2011), pp. 52–78. DOI: 10.4018/978-1-60960-501-8.ch004 (cit. on p. 6).

[10] Jean-Éric Pelet and Panagiota Papadopoulou. 'The effect of colors of e-commerce websites on consumer mood, memorization and buying intention'. In: *European Journal of Information Systems* 21.4 (2012), pp. 438–467. DOI: 10.1057/ejis.2012.17. URL: https://doi.org/10.1057/ejis.2012.17 (cit. on p. 6).

[11] Martin F. Porter. 'An algorithm for suffix stripping'. In: *Program* 40 (1980), pp. 211–218. DOI: 10.1108/00330330610681286. URL: https://pdfs.semanticscholar.org/a651/bb7cc7fc68ece0cc66ab921486d163373385.pdf (cit. on p. 11).

[12] Jialun Qin et al. 'Multilingual Web retrieval: An experiment in English–Chinese business intelligence'. In: *Journal of the American Society for Information Science and Technology* 57.5 (2006), pp. 671–683. DOI: 10.1002/asi.20329. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/asi.20329. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.20329 (cit. on p. 63).

[13] Stephen E Robertson et al. 'Okapi at TREC-3'. In: *Nist Special Publication Sp* 109 (1995), p. 109. URL: http://trec.nist.gov/pubs/trec3/papers/city.ps.gz (cit. on p. 13).

[14] José Sousa, Marco Pereira and Joaquim Arnaldo Martins. 'Improving Browser History using Semantic Information'. In: *ICEIS*. 2012. URL: https://pdfs.semanticscholar.org/f635/d0a2b07079b939f7d93834cd39cc104e43ca.pdf (cit. on p. 6).

[15] Chris Staff and Ian Bugeja. 'Automatic Classification of Web Pages into Bookmark Categories'. In: *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '07. Amsterdam, The Netherlands: Association for Computing Machinery, 2007, pp. 731–732. ISBN: 9781595935977. DOI: 10.1145/1277741.1277881. URL: https://doi.org/10.1145/1277741.1277881 (cit. on p. 3).

[16] Linda Tauscher and Saul Greenberg. 'How People Revisit Web Pages: Empirical Findings and Implications for the Design of History Systems'. In: *Int. J. Hum.-Comput. Stud.* 47 (July 1997), pp. 97–137. DOI: 10.1006/ijhc.1997.0125 (cit. on p. 1).

[17] Hugo Zaragoza et al. 'Microsoft Cambridge at TREC 13: Web and Hard Tracks.' In: Jan. 2004. URL: https://trec.nist.gov/pubs/trec13/papers/microsoft-cambridge.web.hard.pdf (cit. on p. 13).