Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Smart Web Filtering

Bachelor Thesis

Sibylle Jeker

jekers@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Philipp Brandes, Pascal Bissig
Prof. Dr. Roger Wattenhofer

August 22, 2016

Abstract

The internet consists of a range of different undesired information. Besides advertising or spam there is also other content, which specific users do not want to read about. While some do not want to see spoilers about the last episode of Game of Thrones, others may watch the finals of the European Championship one day later and therefore do not want to be informed about the result. Apart from that, there is a lot of annoying and uninteresting information as well, for example news about the private life of some celebrities. We introduce a web filter that takes undesired topics of the user as input and finds related terms using Wikidata. After applying our filtering algorithm, unwanted content is hidden and the user can decide to unhide it, if he wants to see it anyway.

Our system managed to hide nearly 90% of chosen undesired topics of different types on 200 web pages. By only considering topics of specific types the system was able to filter 97% of the unwanted content.

Contents

Chapter 1

Introduction

Nowadays the Internet has a big impact on our daily life. We use it for communication, to look up important information, to get updated about current events and much more. But besides useful content, the appearance of annoying and undesired content is not rare either. For advertising there already exist numerous blocker systems, which are very successful. But while the filtering of this content is the same for all users, there exists undesired content that is very user-specific as well. Especially news pages tend to contain either annoying information or spoiler content, which upsets lots of users. We think that a filtering system which takes any unwanted topic as input and then filters the text content of this topic on all web pages is desired by a lot of users. As it is a very user specific problem, it is important that each user can define his own terms that he wants to hide.

There is a big occurrence of shortcuts in reports, for example the alias **GoT** to report the happenings in the last episode of **Game of Thrones**, or the term **FCB** to inform about the result of a game of **FC Basel**. As we want our spoiler filter to block as much related content of the unwanted topic as possible, one of the most important parts of our filter system is to find related terms of the input topic. With this approach we want to make sure that the filter achieves a good performance.

We define our goal to provide a smart web filter that first of all finds related content to any undesired subject and secondly filters spoilers in all kinds of web pages. In further sections **user-defined spoiler term** is used as generic term for undesired topics the user gives as input to our system.

1.1 Related Work

This thesis follows up on the bachelor thesis **Hide Spoiler** [1] and improves it further. The most important improvement was to find additional spoiler terms that are related to the user-defined spoiler terms and can therefore contribute to a better filter performance. Further, we improved the usability of the spoiler filter and found a solution to filter dynamic content on web pages in addition to

the normal content.

There exist numerous filter systems of different types. Well known are browser-based filters like the browser extension **AdBlock** [2], or email filters to block spam content. Other examples are network-based filters or safety filters offered by search engines, which filter out inappropriate search results. These filters differ fundamentally from our system, since one can clearly identify advertising or spam, whereas our web filter depends heavily on the user. Some user-specific approaches already exist, but they are limited either to the input topics or to the web pages. For example Guo et al. [3] use Latent Dirichlet Allocation to find movie spoilers on IMDB. However this is limited to IMDB and since we want to be able to filter arbitrary topics on any web page, different filter methods are required.

In the next section we provide more insight into advertising filters and email spam filters.

1.1.1 Advertising Filter (Ad Filter)

Ad filters [4] are often used as web proxy or browser extension and some antivirus software can act as ad blocker as well. They block advertising, which can pop-up in a new window or is integrated in a web page as image, video, audio or text content. Ad filters often use whitelisting and blacklisting to control advertisements. While whitelists contain elements that can always be approved and do not contain advertising, blacklists store elements that must be hidden. Compared to the browser extension, web proxies have more freedom from implementation limitations, because they are browser independent. However they have difficulties to filter SSL traffic and not the full web page is available to the filter [5].

1.1.2 Email Spam Filter

The job of spam filters is to detect emails containing spam out of all incoming emails of an account. The filters take an email as input and then decide whether it can be moved to the inbox. The difficulty is, that it is much worse to falsely mark an email as spam, than to accidentally permit an email that contains spam in the inbox.

There exist many approaches to implement spam filters. There are list-based filter systems [6], which use lists (e.g. whitelists or blacklists) to find out if the user sending the email can be trusted or not. Alternatively one can use content-analysis [7] by analyzing words or phrases that are contained in the email. More advanced are Bayesian Filters [8], which are trained by user inputs and use machine learning to calculate the probability of spam content in each email.

Chapter 2

Background

2.1 Wikidata

To find related content to the user-defined spoiler terms, we queried the database of Wikidata [9]. Wikidata is a large knowledge database, that provides structured data. The data consists mainly of different items with unique identifiers, that represent a topic. The items contain optional labels, descriptions and aliases in different languages and several statements. A statement consists of an optional reference and a claim, where the claim contains a property, a value and an optional qualifier. All properties consist of unique identifiers as well. The value can be another item or other data types (e.g. date, time, coordinates). When the value of a statement is another item, the item is linked to the value.

Figure 2.1 represents a simplified Wikidata entry of **Game of Thrones**. It contains a label, an identifier, a description and aliases. The statement contains a reference, collapsed to **>reference 1** and a claim. The property of the claim is **cast member** and the value is **Kit Harington**. The claim also contains a qualifier **character role** with the value **Jon Snow**.

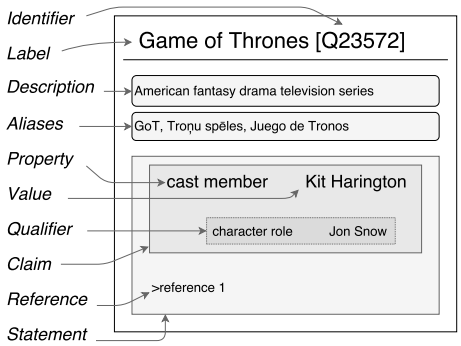


Figure 2.1: Simplified representation of the Wikidata entry of **Game of Thrones**.

Chapter 3

Design

3.1 Spoiler Terms

One of the most important parts of the spoiler filter are the spoiler terms. The spoiler terms help to hide content on web pages, the user does not want to read about. The first part of spoiler terms is defined by the user, we call them user-defined spoiler terms. If the user wants to, he can link his terms to the corresponding Wikidata items. This allows us to find additional spoiler terms, which are related to the user-defined spoiler term by querying the Wikidata dataset.

We define the set of spoiler terms as the union of the user-defined spoiler terms and the corresponding list of additional spoiler terms. All duplicated terms and terms that contain other terms are removed from the set. This means, if the spoiler term set for example contains the values **Federer** and **Roger Federer**, the term **Roger Federer** is removed, because every report that contains **Roger Federer** also contains the term **Federer**. By removing those terms, the algorithm achieves a better run-time performance, but filters the same spoiler content.

The next section describes the approach to generate additional spoiler terms for the user-defined spoiler terms.

3.1.1 Generation of Additional Spoiler Terms

The first idea to query additional spoiler terms was to request all incoming and outgoing nodes from the Wikidata item of the user-defined spoiler term and filter the returned items. We analyzed the sum of incoming and outgoing nodes of the returned items and thereby wanted to determine the importance of them. It turned out to be more difficult than assumed to find a parallelism in this sum of different items. Especially geographical items such as **USA** appeared quite often with a big divergence of the mentioned sum. Therefore, we did not find a way to determine the importance of the returned items and dropped this idea.

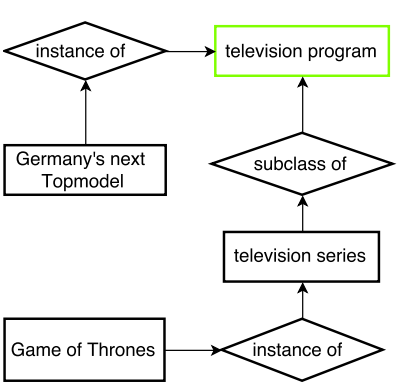
Figure 3.1: Wikidata entry of **Game of Thrones** [10]. The aliases are listed in the column **Also known as**.

The second approach was to query the Wikdidata dataset with more precise queries. We implemented four different queries to find additional spoiler terms. The first is a general query, which we applied to all user-defined terms.

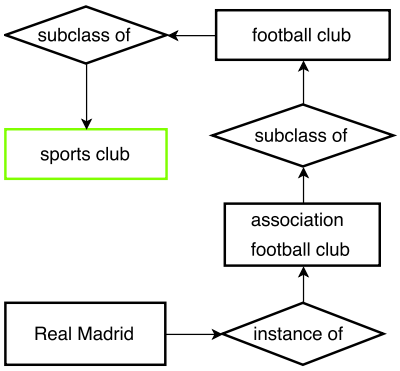
As Wikidata provides a list with aliases of all items, we queried those aliases and added them to our additional spoiler term list. The aliases are defined in different languages. In Figure 3.1 we can see the first section of the Wikidata entry of **Game of Thrones**. The important alias **GoT** is not listed in all languages and therefore, the **Game of Thrones** item is not complete. This is the case for a lot of items and the reason we decided to add the aliases of all languages with latin alphabet to the additional spoiler list. Thereby we assure not to miss any important alias that is given in at least one language.

Some Wikidata item labels start with an article. Whenever this was the case for a user-defined spoiler term, we added an additional spoiler term with the same content, but removed the article. For example the names of the television shows **The Bachelorette** or **The Big Bang Theory** start with an article. As a report about the series does not necessarily always use an article before the actual term, it is better to filter the report with the terms **Bachelorette** and **Big Bang Theory**.

We formed three more queries, which we only requested for user-defined spoiler terms of a particular type. The first group belonged to user-defined spoiler terms that are people, to hide for example information about certain famous people, elections or individual athletes. Because the tendency of spoilers in television series or films and sports is very high, we secondly focused on those two groups of items. We labeled these three groups **People**, **Sports** and **TV**. To find out, of which type the user-defined spoilers are, we used the properties **instance of** and **subclass** that Wikidata provides. We requested those properties recursively for the Wikidata item of the user-defined spoiler term and compared them to



(a) Example of two items, which we grouped into the **TV** class.



(b) Example of the Wikidata item **Real Madrid**, which we grouped into the **Sports** class.

Figure 3.2: Two examples to illustrate how we grouped the items into the different classes.

the items **human** for the group **People**, **sports club** and **sports team** for the group **Sports** and **television program** and **film** for the group **TV**. As we assigned each user-defined spoiler to at most one group, we stopped the recursion as soon as we found a match.

In Figure 3.2 we show examples of how we grouped certain items into the groups **TV** and **Sports**. The first Example 3.2a is a simplified graph of the items of the TV series **Game of Thrones** and the TV show **Germany's next Topmodel** and their recursive request of the properties **instance of** and **subclass of**. While the property **instance of** of **Germany's next Topmodel** is **television program** and **Germany's next Topmodel** is added to the **TV** class directly, we have to apply recursion to find, that **television series** is a subclass of **television program** and **Game of Thrones** can be added to the **TV** class as well.

The second Example 3.2b illustrates the recursive request of the properties **instance of** and **subclass of** of the Wikidata item **Real Madrid**. After querying the **subclass of** property on **association football club** twice, we discover that **Real Madrid** belongs to class **Sports**, because the queries returned the item **sports club**.

The next section describes the queries of the three groups in detail.

People

If the user-defined term is a human, it is advantageous to add the last name to the additional terms. When a writer refers to a person, he often only uses the person's family name. To query the family name, we first requested the

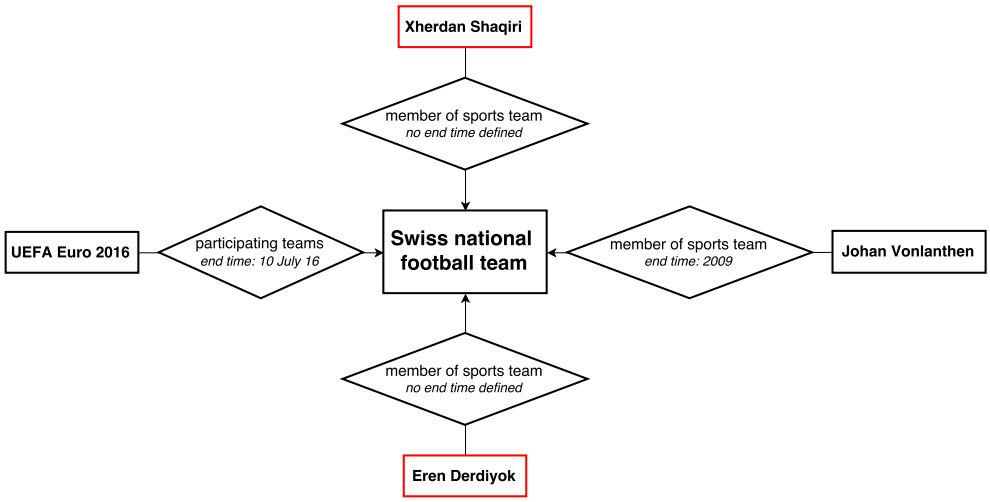


Figure 3.3: A simplified example of the Wikidata item **Swiss national football team** to illustrate the queries of the **Sports** group. The items in red are added to the additional spoiler list.

family name from the Wikidata database with the property **family name**. If the Wikidata item of the user-defined spoiler term was incomplete and did not contain a claim with the property **family name**, we parsed the last word of the user-defined term manually and assumed, it matched the correct last name.

Sports

To filter content that is related to a specific sports team, we first queried all members of that sports team. We achieved this by first requesting a list of items that contained a claim with property **member of sports team** and the user-defined item as value. Because we only wanted to consider current members, we filtered the items, whose **member of sports team** statement contained a qualifier **end time**.

Secondly, we found out in which tournaments the sports team was currently participating. To achieve that, we queried all Wikidata items, which contained a claim with property **participating teams** and the user-defined item as value. To filter out tournaments that already took place in the past, we requested the **end time** property of each tournament and compared it to the date on which the user defined his spoiler term for the first time. If the value of the **end time** property was an earlier date than the date of definition, the tournament got filtered out.

Figure 3.3 shows an example of the item **Swiss national football team**. The items **Xherdan Shaqiri** and **Eren Derdiyok**, which are highlighted in red, both contain a claim with property **member of sports team** without a qualifier

end time and are therefore added to the additional spoiler list. The item **Johan Vonlanthen** on the right hand side is not a current member, because the claim contains a qualifier with property **end time** and is therefore not added to the spoiler list. On the left hand side is the tournament item **UEFA Euro 2016**, which contains a claim with property **participating teams** with the value **Swiss national football team**, but the value of **end time** is in the past, which is why it is not added to the additional spoiler list. We added for all additional spoiler terms the aliases and for humans the last names to the additional spoiler list, in case Wikidata provided them.

TV

To get additional terms of television programs or films, we wanted to find all characters and cast members. Because many Wikidata items do not contain complete claims with property **cast member** and qualifier **character role**, we applied a more general approach. Our query first requests all outgoing claims with maximum distance 1 and all incoming claims with maximum distance 2 of the Wikdiata item of the user-defined spoiler term. Because we are only interested in real humans or fictional characters, we filtered out all the other claims not belonging to these categories. To filter the list, we used the **instance of** property and compared its value to the items **fictional human**, **fictional character** and **human**.

Figure 3.4 illustrates a simplified example of this query. The Wikidata items **USA**, **David Beniott** and **Kit Harington** are values of outgoing claims of the item **Game of Thrones** with distance 1. While **USA** is of instance **country** and therefore not added to the additional spoiler list, **Kit Harington** and **David Beniott** are of instance **human** and added to the list. Another example is the Wikidata item of the Game of Thrones character **Jon Snow**. It is linked to the item **Game of Thrones** with distance 2. Because it is of instance **fictional human**, is is added to the additional spoiler list. We added for all additional spoiler terms the aliases, for humans the last names and for characters the last- and given name to the additional spoiler list, in case Wikidata provided them.

3.2 Filtering System

This section explains the filtering system we applied to search and hide spoiler content on web pages. The algorithm is based on a former Bachelor Thesis [1]. It accesses and changes the HTML Document Object Model (DOM) [11] tree of web pages, which contains different objects and is created by the browser when a web page loads. Figure 3.5 shows an example of a DOM tree in HTML structure and Figure 3.6 shows the same document in tree representation. The document abstracts a web page that contains spoiler content in three nodes (3, 7, 10). The

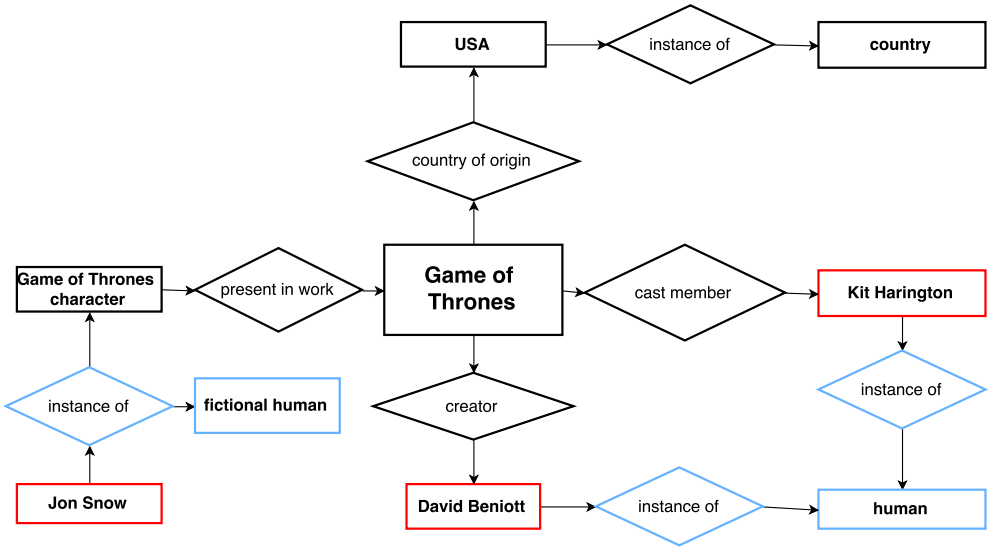


Figure 3.4: Simplified example of the additional spoiler term search of **Game of Thrones** of the class **TV**. The items highlighted in red are the detected additional terms.

```
<html>
<head> </head>
<body>
   <div> 1: No spoiler
      <div> 2: No spoiler </div>
      <div> 3: Spoiler
         <div> 4: No spoiler </div>
         <div> 5: No spoiler </div>
      </div>
   </div>
   <div> 6: No spoiler
      <div> 7: Spoiler </div>
      <div> 8: No spoiler </div>
      <div> 9: No spoiler
         <div> 10: Spoiler </div>
      </div>
   </div>
   <div> 11: No spoiler </div>
</body>
</html>
```

Figure 3.5: Simplified HTML document with some spoiler content to illustrate how the sub-tree of the body node can be structured.

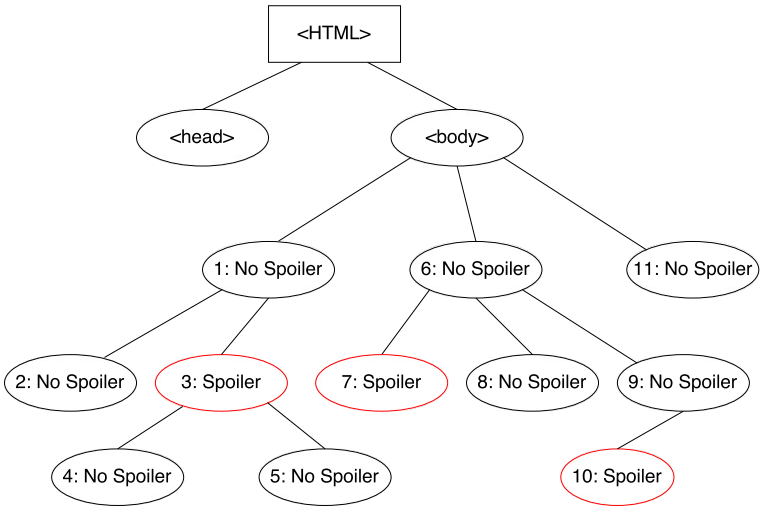


Figure 3.6: DOM tree of the HTML document in Figure 3.5.

next sections will follow this example document closely to illustrate the filter algorithm in detail. Because the head node only contains meta information and scripts, we do not consider it and leave it out in the next sections.

3.2.1 Overview

The main approach was to filter not just the spoiler terms, but also related content to those spoiler terms. For example, when a user defines **Game of Thrones** as a spoiler term, he probably wants whole articles about **Game of Thrones** episodes hidden and not just the term **Game of Thrones** in it. To achieve that, an obvious solution is to hide the whole content of the web page. As this is clearly not a good approach, our algorithm must find a way to hide the content at the right DOM node. According to these thoughts, we defined two equally important goals of our filter system as follows:

1. Hide all content that contains a spoiler term or spoiler related information.
2. Do not hide any content, which does not contain any spoiler term or spoiler related information.

The algorithm traverses the DOM tree three times. The first two traversals are used for spoiler detection and the third one to finally cut the tree and filter the spoilers. The next two sections explain these traversals in detail.

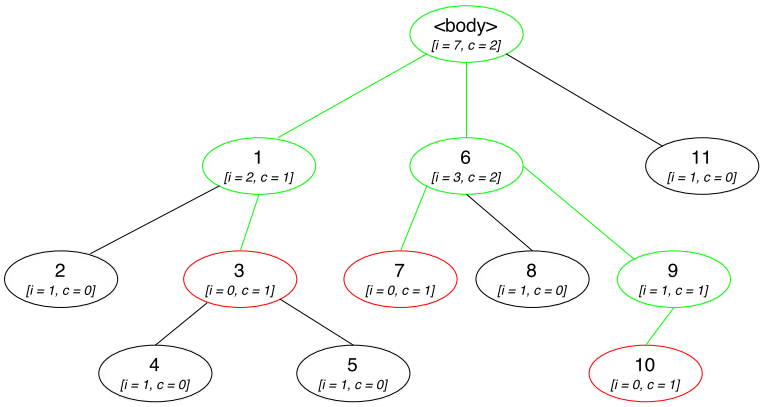


Figure 3.7: DOM tree with the spoiler content in red and the paths containing spoilers in green. For every node we calculated the innocent length i and the number of cuts c.

3.2.2 Spoiler Detection

To detect spoilers the algorithm traverses the DOM tree from the body node to the leaf nodes. To achieve a good run-time performance, it only traverses the tree where actual spoiler content occurs. It makes a depth-first traversal and stores in every node the number of spoilers that occur in the node or the sub-tree of the node. If the spoiler count is greater than zero, it continues the search for all children nodes.

In Figure 3.7 the three green highlighted paths (<body>, 1, 3), (<body>, 6, 7) and (<body>, 6, 9, 10) are containing spoilers and traversed by the algorithm. The path (<body>, 1, 3) stops at node 3, because node 3 contains a spoiler. The inner node 2 or his sub-tree does not contain spoiler content and is therefore not included in a spoiler path.

Now we must decide, where in the path it is best to cut the tree and therefore hide the nodes containing spoilers. For this purpose we traverse the same paths from the bottom to the top and calculate two new parameters for every node. The first parameter is the number of cuts c we have to perform at each node, to hide all children nodes with spoiler content. In the example of Figure 3.7, we must perform two cuts at the body node (nodes 1 and 6) to hide all spoilers. An alternative solution is to cut the spoiler nodes at node 1, node 6 and node 9, which would result in a total of 3 cuts.

The second parameter is the innocent length i. It is defined as the length of the text content that does not contain any spoilers. The innocent length i of non-leaf nodes sums up the innocent length of all children nodes and the node's text length. If a node contains spoilers, we define its innocent length as 0. We simplified the example of Figure 3.7, by defining the text length of all nodes as

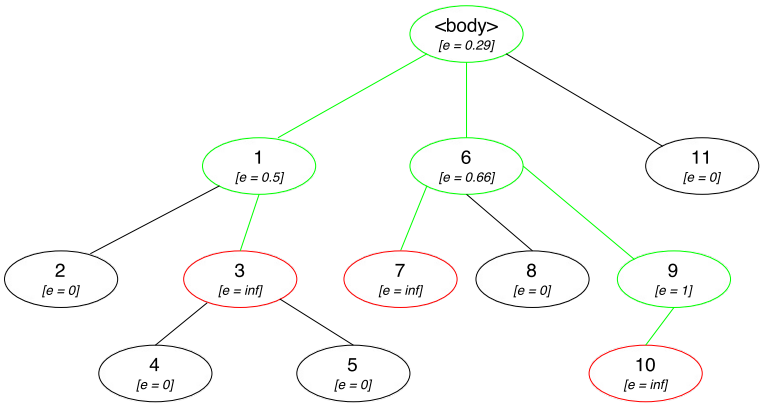


Figure 3.8: DOM tree with the calculated value of e of every node.

1. While the innocent text of all red highlighted spoiler nodes is 0, the innocent length of the non-leaf node 6 sums up all innocent lengths of his children nodes (7, 8, 9) and then adds its own text length to it, which results in the sum $0 + 1 + 1 + 1 = 3$.

3.2.3 Spoiler Filtering

The parameter i shows how much innocent content a node has. The greater the value of i is, the less likely the node should be hidden, because the goal is to hide as little innocent content as possible. This means that we want to perform the cut such that the sum of innocent text is minimized, but we also have to take the value of c into account.

As an example we imagine a tree that contains 4 sibling nodes and three of them contain spoilers. Now the question is, if it is better to cut the parent of those nodes and therefore hide the fourth non-spoiler node as well, or perform 3 independent cuts. It depends on the value of i, which approach is better. If the value of i of the fourth node is great, it may be better not to hide it, because we would hide a lot of innocent text. When the value of i is small, the probability is bigger, that the fourth node's text content is related to the text content of the other nodes and therefore should be hidden as well. This is for example the case when the fourth node contains the title of an article whose remaining text content is contained in the other three nodes. To cut the DOM tree at the right position we want to find the optimal trade-off between the values c and i and define the threshold e as follows:

$$e = \frac{c}{i}$$

The algorithm calculates e for every node in the last traversal of the paths

containing spoilers from the root node to the leaf nodes. When the value of e is greater than a predefined threshold, we perform the cut to hide the node and all sub-nodes. To make sure that nodes with spoiler content are always hidden, we define the threshold e of these nodes as infinity.

In Figure 3.8 e is calculated for every node. We can see that in each spoiler-path, which is highlighted in green, the value of e increases while traversing from the top to the bottom of the tree. Somewhere in this path the value of e must be bigger than the predefined threshold, as the last node is always set to infinity. The algorithm therefore finds the optimal node somewhere in the path and performs the cut. The predefined threshold is determined in the Evaluation 5.1.

Chapter 4

Implementation

We chose to implement the web filter as a Google Chrome extension mainly because the extension has direct access to the web page's content and Chrome provides several very useful Javascript APIs [12]. Another reason was that Google Chrome is currently the most used web browser [13, 14] and therefore, our tool can easily be used by many people.

The Chrome extension contains three main scripts:

1. **The background script** is an invisible event page and is executed when it receives a message from one of the other scripts.
2. **The popup script** contains a web page that is used as a pop-up window, where the user can define his preferences.
3. **The content script** can read and modify the content of web pages. Every tab in Google Chrome runs its own content script. It has no access to the background or popup script and can only communicate with them via storage or messages.

In Figure 4.1 we show the main tasks of each script and interactions with other scripts. The starting points are highlighted in green and the conditions are marked yellow. We can see that the background script gets invoked by the other scripts and does not run on its own. The popup script is executed as soon as the DOM tree of the pop-up window is loaded (**1**). The content script runs first when the DOM tree of the web page starts loading (**2**) and can then be invoked by the two events **DOMContentLoaded** (**3**) and **DOMNodeInserted** (**4**) as well. We will now explain the execution of each starting point to illustrate the tasks of the Chrome extension in more detail.

The first starting point is in the popup script, whose task is to receive and store the preferences (Figure 4.2) the user wants to define. For this purpose, the script stores all preferences to memory as soon as the user submits them. Searching the additional spoiler terms cannot be done by the popup script, because it

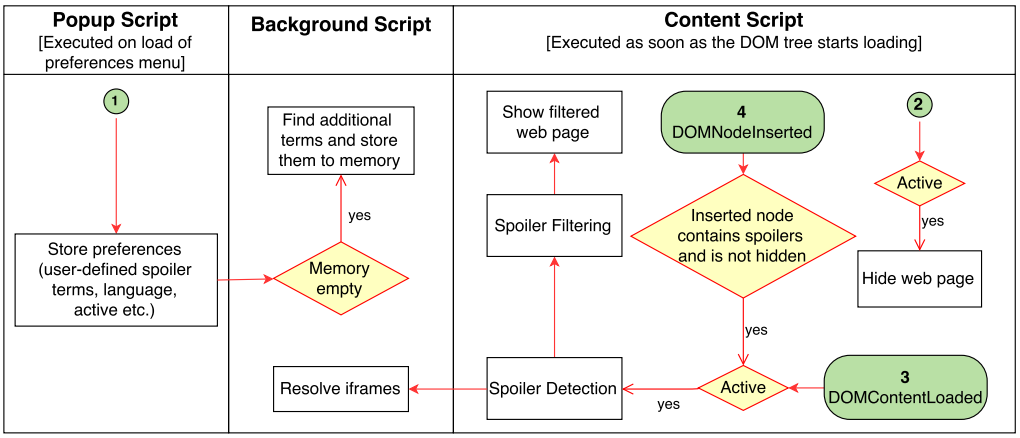


Figure 4.1: Simplified representation of the main tasks and interactions of the popup, background and content scripts in the Google Chrome extension.

runs only as long as the pop-up window is opened. This is why the script sends a message to the background script, which says that it should search additional spoiler terms for the new user-defined spoiler terms. The background script first checks, if the additional spoiler terms are already stored in memory. If they are not, it requests them and stores them.

Our goal was to hide the whole web page until we filtered all spoilers of the web page. Otherwise spoilers on top of the web page could be visible for a very short time and seen by the user. Because the starting point **2** is executed when the DOM tree starts loading, the web page is not yet visible at that time. Therefore, it is the best moment to hide the web page until the algorithm finished filtering the spoilers. We only hide the web page if the user set the variable **Active** to true. **Active** defines if the spoiler filter runs for every page automatically and is set to true by default.

The main part of the filter algorithm is executed after the starting point **3**, when the event **DomContentLoaded** is fired. At this point we have access to the full DOM tree and can therefore start the tree traversals. If the **Active** variable is set to true it first executes the **Spoiler Detection**, then the **Spoiler Filtering** and at the end it makes the web page visible again. Resolving inline-frames, is done by the background script, which gets notified by the content script as soon as the URLs of the inline-frames are stored to memory. We explain this further in Section 4.4.1.

Starting point **4** is executed when the **DOMNodeInserted** event is fired. This event notifies the script as soon as any node is inserted to the DOM tree. The purpose of this starting point is to fetch dynamic content, which is loaded after the **DOMContentLoaded** event. If the inserted node contains a spoiler term, is not hidden yet and **Active** is set to true, the content script filters the

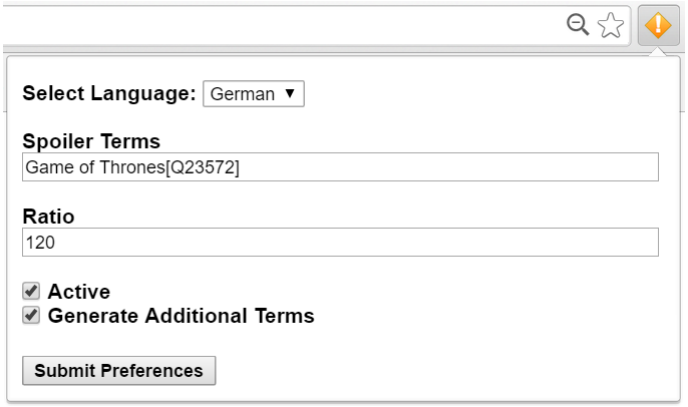


Figure 4.2: Toolbar button with the preferences menu of the Google Chrome extension.

inserted node in the same way it filtered the whole DOM tree in the execution of starting point **3**.

4.1 Graphical User Interface (GUI)

The GUI consists of a toolbar button and the preferences menu. The preferences menu can be accessed by clicking on the toolbar button and has the purpose to define different parameters. The user can choose between the languages **German** and **English** for the search of additional terms with Wikidata, which we described in Section 3.1.1. He can also enter, whether he wants to generate additional terms with the value of **Generate additional spoiler terms** and if the extension is active with the value of **Active**.

The user can define all spoiler terms in the text-box **Spoiler Terms**. While entering the spoiler terms, a drop-down menu with corresponding Wikidata entries appears. When the user clicks on a Wikidata item, the identifier of the Wikidata item is stored in addition to the spoiler term and can therefore be used to search additional spoiler terms. In Figure 4.2 we can see that next to the spoiler term **Game of Thrones** the identifier **Q23572** is stored. If the user does not click on a Wikidata entry no additional terms are generated. In the text-box **Ratio** the user can enter the value of the threshold e, which we defined in Section 3.2.3. The default value is set according to the results of the Evaluation 5.1. As soon as all preferences are set, the user can store them and close the menu by clicking **Submit Preferences**.

4.2 Generation of Additional Spoiler Terms

4.2.1 Wikidata Query Service

The Wikidata Query Service [15] allows us to run complex queries on the Wikidata knowledge graph. The data is exported as Resource Description Framework (RDF) dump, which is basically a list of subject-predicate-object triples. To provide fast access, the RDF dump is preprocessed and finally imported to a Blazegraph database. The data can be accessed by SPARQL [16] queries, which we defined in the background script according to the queries we resolved in Section 3.1.1. The background script accesses the Wikidata Query Service with XMLHttpRequests [17] to request additional spoiler terms of the Wikidata dataset and parses the received data from JSON format to additional spoiler term labels.

4.3 Storage and Communication

We used the asynchronous scripts **onMessage** and **sendMessage**, which are provided by the **chrome.runtime API** [18] to receive and send messages. To store the preferences of the user, the additional spoiler terms and different evaluation data we used the **chrome.storage API** [19].

4.4 Spoiler Detection

In the first depth-first traversal of the DOM tree, we stored in every node element an attribute **nodeIndex**, which we used in the further steps of the algorithm to identify each node uniquely. To hide the correct nodes and to compute the parameters e, i and c of each node, we counted the occurrences of spoiler terms in each node. Therefore, we requested the text content of each node and searched it with regular expressions. The text content of each node does also contain the text content of its sub-tree. This simplified finding the paths containing spoilers of the DOM tree.

The image nodes do not contain text content, but an **alt** attribute, whose text content is a description of the image and a **src** attribute, which contains the URL of the image. To filter the image we requested these attributes of each image node separately. Because we only traversed the DOM tree where actual spoiler content occurred, we requested at every node we visited all sub-nodes that were images. Then we searched these images separately for spoiler content.

4.4.1 Dynamic Content

We distinguished between two dynamic content classes. The first ones are the inline-frames (iframes), which are basically other HTML documents placed into a web page. They contain a URL or path with the corresponding HTML document and are often used for live tickers. To find spoiler content in iframes, we requested the DOM tree of the corresponding URL with an XMLHttpRequest and applied the filter algorithm to it. If somewhere in the iframe's DOM tree a spoiler term was found, we hided the iframe.

The second dynamic content is inserted to the DOM tree by script nodes after the DOMContentLoaded event is fired. Many news pages load new content continuously when the user scrolls to the bottom of the page. By explaining Figure 4.1 we already showed how we used the event **DOMNodeInserted** to find spoilers in such dynamic loaded web pages.

4.5 Spoiler Filtering

As soon as we found the right nodes to hide we set the visibility of those nodes as hidden and inserted a spoiler button as sibling node. The spoiler button toggles the visibility of the hidden node whenever it is clicked.

Chapter 5

Evaluation

To evaluate the spoiler filter and the choice of the additional spoiler terms, we compared the performance of the filter with and without additional spoiler terms. We tested 10 different news web pages with different spoiler terms daily for 20 days. This is a total of 200 web pages we evaluated. We downloaded the web pages from 20th April 2016 until 9th May 2016 to analyze the pages offline. Table 5.1 lists the detailed spoiler terms and web pages. To evaluate the web pages we manually determined for each web page and spoiler term the parts that should be hidden and stored the corresponding **nodeIndex** of the node with the character text length. During the evaluation we compared this information with the web page filtered by the system.

News Page	**Spoiler Terms**
srf.ch	Prince
handelszeitung.ch	Volkswagen, Mitsubishi
20min.ch	Game of Thrones, Die Bachelorette
bernerzeitung.ch	Donald Trump
tagesanzeiger.ch	Roger Federer, Wladimir Putin
blick.ch	Simonetta Sommaruga
news.google.ch	Hillary Clinton, Nashville Predators
swissinfo.ch	Katastrophe von Tschernobyl
nzz.ch	FC Basel
news.ch	Cristiano Ronaldo

Table 5.1: List of web pages with the corresponding spoiler terms we used to evaluate the spoiler filter.

We used the Receiver Operating Characteristic (ROC) [20] curve to plot the results of our tests and to analyze it. To construct a ROC curve, a few parameters need to be computed in advance:

- **True Positive (TP):** True positive is the number of characters that were correctly hidden.

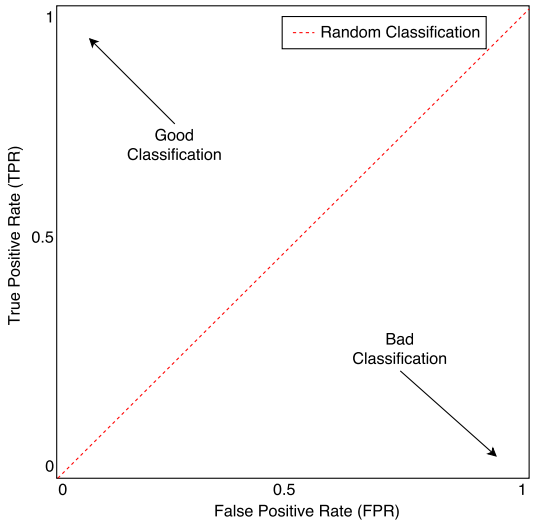


Figure 5.1: ROC curve interpretations.

- **True Negative (TN):** True negative is the number of characters that were correctly not hidden.
- **False Positive (FP):** False positive is the number of characters that should not be hidden, but the filter system hided them by mistake.
- **False Negative (FN):** False negative is the number of characters that should be hidden, but the filter system did not hide them.
- **True Positive Rate (TPR):** The true positive rate is computed with the following formula:

$$TPR = \frac{TP}{TP + FN}$$

- **False Positive Rate (FPR):** The false positive rate is computed with the following formula:

$$FPR = \frac{FP}{FP + TN}$$

The ROC curve plots the FPR (x-axis) against the TPR (y-axis). As we wanted to find the optimal value of the threshold e, we computed the FPR and TPR for different e, which results in a curve. Figure 5.1 illustrates how to interpret the ROC curve. The red line in the middle is the result of a random classification, since the number of hits and misses is the same. As a perfect classification is in the upper left corner, we choose the value of e with minimal euclidean distance to this corner to be the optimal threshold.

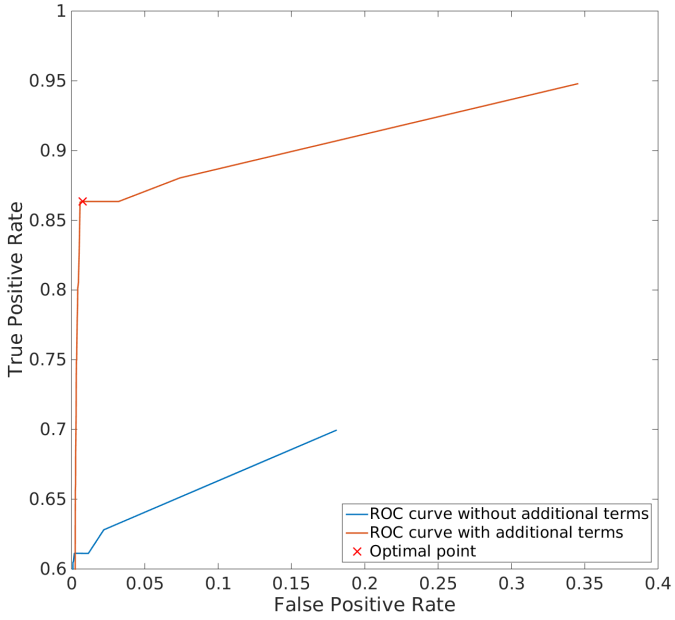


Figure 5.2: ROC curves with and without additional terms with the threshold e in range from 0 to 0.004.

5.1 Results

We computed the TPR and FPR for e in the range from 0 to 0.004 of the spoiler filter with and without additional spoiler terms. Figure 5.2 shows the resulting ROC curves. The lines start on the right where e is equal to zero and end in the bottom left corner. A point close to the y-axis means, that it has a low FPR and the number of non-spoiler characters that were wrongly hidden is therefore close to zero. When the point is located near the line where y is equal to 1, it means that it has a high TPR and has therefore hidden most of the spoiler content. The reason why the lines do not reach the top where the TPR is equal to 1 is that when the filter finds no spoiler term on the web page, nothing gets hidden no matter how small the value of e is. The optimal point (0.0075,0.8634) has the smallest euclidean distance to the point (0,1) and is achieved with additional spoiler terms. The coordinates of the optimal point state that the filter with additional spoiler terms hides approximately 86% of the spoiler content and 0.75% of the non-spoiler content. When we compare the two curves, we can say that the upper curve nearly doubles the true positive rate of the lower curve. Therefore, the filter hides almost 30% more spoiler content by using additional spoiler terms. On the other hand, the filter hides 0.05% more non-spoiler content.

Because normally just a small part of a web page contains spoilers, the value of TN is compared to the values of TP, FP and FN much greater. The average value of true negatives is 58470, while the other average values are 247(TP), 345 (FP) and 39 (FN). Therefore, our web filter achieves a very good FPR compared to the TPR.

The next section shows the results of each web page separately with the value of e equal to the optimal threshold. As we look at the results of a particular e, the plots contain points and not curves anymore. To evaluate the choice of the additional spoiler terms, we plotted the performance of the filter without additional spoiler terms (blue) together with the performance of the filter with additional spoiler terms (green) on Figure 5.3. To be able to analyze the results more in detail, we evaluated the web pages **20min.ch** and **news.google.ch** for each spoiler term separately.

We can see that the points **1, 10** and **12** achieved the same performance with and without additional spoiler terms (red). Point **1** with the user-defined spoiler term **Prince** can not be improved, because the TPR is equal to 1, which means that every article with spoilers contained the term **Prince** and was therefore hidden correctly. Regarding the other two points, the spoiler filter did not find meaningful additional spoiler terms for the user-defined spoiler terms **Katastrophe von Tschernobyl** and **Cristiano Ronaldo**. The Wikidata entry of **Katastrophe von Tschernobyl** does not contain useful aliases and because the articles about this topic in the news page **swissinfo.ch** not always contain the whole term **Katastrophe von Tschernobyl**, there was a lot of spoiler content, that was not correctly hidden. The problem with the spoiler term **Cristiano Ronaldo** is related to the absence of Wikidata entries as well. Much articles refer to **Cristiano Ronaldo** by just using his second given name **Ronaldo**. As we mentioned in Section 3.1.1, we parsed the last word of the user-defined spoiler term manually only if there is no claim defined with property **family name**. Because the Wikidata entry of **Cristiano Ronaldo** does refer to his family name and his middle name is not contained in the aliases list, the filter misses some important spoilers.

Point **11** with corresponding spoiler term **FC Basel** achieved a much better TPR with additional spoilers, especially because of the additional spoiler alias **FCB**, which was used in a lot of articles. When a report about a previous game just contained for example the phrases **Basel 1:0 Vaduz** or **Wann wird Basel Meister?**, the filter had problems finding the spoiler, because the term **Basel** was not contained in the additional spoiler terms list. Another noticeable problem is that the false positive rate has changed for the worse and therefore the filter has hidden too much content. The reason for this are the last names of the association football players. For example the last names of the players **Eduard Bauer** or **Danique Stein** can also be used in other contexts, therefore too much content gets hidden. Because Wikidata contains a lot of incomplete items, some

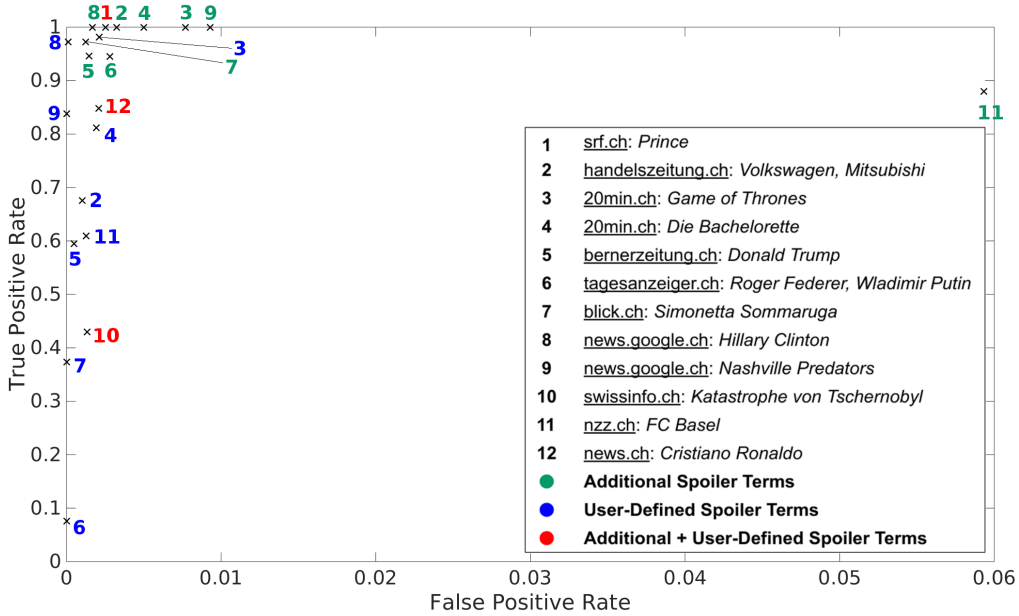


Figure 5.3: ROC points of the evaluation with additional terms (green) and without additional terms (blue). The numbers in red achieved the same result in both tests.

players, which do not play for **FC Basel** anymore still do not contain an **end time** property and therefore many outdated additional terms were found.

By looking at the other points, we can see that the remaining web pages got filtered a lot better with additional spoiler terms. For example adding the alias **VW** of the spoiler term **Volkswagen**, the last names of humans or the term **Bachelorette** for the user-defined spoiler term **Die Bachelorette** to the additional spoiler list resulted in much better filtered web pages.

Obviously the user-defined spoiler term **Katastrophe von Tschernobyl** performs by far the worst. As historical event it is a completely different spoiler term than the others, which are humans, companies, TV shows or sports teams. Because users can not be spoiled about a passed event, it is rather not a topic that is chosen to be hidden by a lot of people. This is why we plotted an additional ROC curve without the web page **swissinfo.ch** with the spoiler term **Katastrophe von Tschernobyl** on Figure 5.4. The new optimal point achieves a much better true positive rate than the optimal point of the ROC curves with all test sets. Without the web page **swissinfo.ch**, the filter system hides about 97% of the spoiler content in the web pages. The conclusion of this observation is that it is very important to choose the correct user-defined spoiler term. It is very likely that the spoiler filter would have achieved a better performance with the user-defined spoiler term **Tschernobyl** instead of **Katastrophe von**

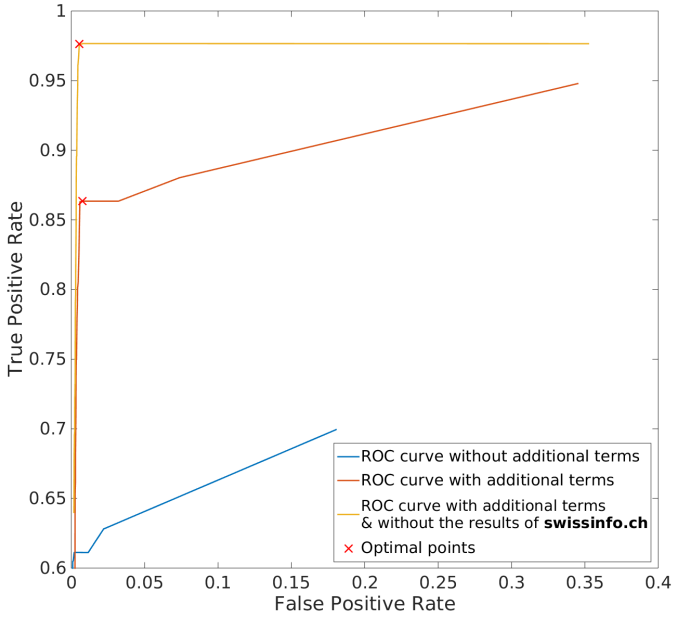


Figure 5.4: The ROC curve with additional spoiler terms and without the test set of the web page **swissinfo.ch** in addition to the curves with all test sets.

Tschernobyl. As the new optimal point (0.0055,0.9762) is reached with the value of e equal to 0.0012, we define the optimal threshold eo as 0.0012.

Chapter 6

Conclusion and Future Work

We implemented a web filter that finds related content of spoiler terms and is able to hide most of the spoiler content on web pages. While it is optimized for the three categories **People**, **Sports** and **TV**, it is also able to find meaningful additional spoiler terms in other categories, given that aliases are defined in the corresponding Wikidata items. Therefore the performance of the filter depends on the choice of the correct Wikidata item and on the completeness of the corresponding Wikidata item. If important aliases are not mentioned in the Wikidata item of a spoiler topic, it is possible that the filter is unable to hide some related content. In the next section we suggest some approaches, which can be used to improve the spoiler filter in the future.

The first improvement addresses the search of additional spoiler terms. We mentioned before that the incompleteness of Wikidata affects the performance of the filter a lot. To bypass this issue, one could query additional databases to receive related content. As the queries are currently optimized for three groups, the search can be improved by either adding more queries for specific groups or by changing them to a more general query which would probably result in better additional spoiler terms for user-defined spoiler terms that are not instance of such a group.

We provided a solution to fetch dynamic content on web pages, which is added by scripts. Even though our solution works in practise, there may be a way to improve it further by not applying the filtering algorithm to the inserted node, but to the body node. This approach would avoid the insertion of too many spoiler buttons, because the system would reapply the filtering algorithm to the whole page every time new spoiler content is loaded, but it is difficult to still achieve a good run-time performance.

So far we hided nodes when they contained at least one spoiler term. As improvement one can define **combined spoiler terms**, which must occur together in a text node so that the corresponding node will be hidden. For example when the terms **Basel** and **Fussball** both appear in a text node, it is likely that the node involves content about **FC Basel**. To define these two terms as normal additional terms of the user-defined spoiler term **FC Basel** would result in too

much hidden content, because both of them can be used in many other contexts as well.

Another interesting approach is to provide a feedback system for the user, which allows to mark either wrongly hidden content or spoiler content that was not hidden. With this additional information the spoiler filter can adapt the choice of the additional spoiler terms to the user and improve its performance over and over. When we assume that several users want to filter the same topics, the filter system could even reuse the collected data collaboratively for different users.

Bibliography

[1] Willi, R.: Hide spoiler. Bachelor thesis, ETH Zurich (2014)

[2] AdBlock. https://getadblock.com/ Accessed on 2016-08-19.

[3] Guo, S., Ramakrishnan, N.: Finding the storyteller: Automatic spoiler tagging using linguistic cues. In: COLING 2010, 23rd International Conference on Computational Linguistics, Proceedings of the Conference, 23-27 August 2010, Beijing, China. (2010) 412–420

[4] Pomeranz, H.: A simple dns-based approach for blocking web advertising. (2013)

[5] Wikipedia: Ad blocking, external programs. https://en.wikipedia.org/wiki/Ad_blocking#External_programs Accessed on 2016-08-19.

[6] Dietrich, C.J., Rossow, C.: Empirical research of IP blacklists. In: ISSE 2008 - Securing Electronic Busines Processes, Highlights of the Information Security Solutions Europe 2008 Conference, 7-9 October 2008, Madrid, Spain. (2008) 163–171

[7] Alexandros Ntoulas, Marc Najork, M.M.D.F.: Detecting spam web pages through content analysis. In: 15th International World Wide Web Conference (WWW), Edinburgh, Scotland, Association for Computing Machinery, Inc. (May 2006)

[8] Sahami, M., Dumais, S., Heckerman, D., Horvitz, E.: A bayesian approach to filtering junk E-mail. In: Learning for Text Categorization: Papers from the 1998 Workshop, Madison, Wisconsin, AAAI Technical Report WS-98-05 (1998)

[9] Wikimedia-Foundation: Wikidata. https://www.wikidata.org/ Accessed on 2016-08-04.

[10] Wikidata: Game of thrones. https://www.wikidata.org/wiki/Q23572 Accessed on 2016-08-04.

[11] Mike Champion, Steve Byrne, G.N.L.W.: Document object model (core) level 1. https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html/ Accessed on 2016-08-05.

[12] Chrome, G.: Javascript apis. https://developer.chrome.com/extensions/api_index/ Accessed on 2016-08-06.

[13] StatCounter: Top 5 desktop, tablet & console browsers. http://gs.statcounter.com/ Accessed on 2016-08-04.

[14] W3Counter: Web browser market share. https://www.w3counter.com/globalstats.php/ Accessed on 2016-08-04.

[15] Wikimedia-Foundation: Wikidata query service. https://query.wikidata.org/ Accessed on 2016-08-04.

[16] Eric Prud'hommeaux, A.S.: Sparql query language for rdf. https://www.w3.org/TR/rdf-sparql-query/ Accessed on 2016-08-04.

[17] Anne van Kesteren, Julian Aubourg, J.S.H.R.M.S.: Xmlhttprequest level 1. https://www.w3.org/TR/XMLHttpRequest/ Accessed on 2016-08-04.

[18] Google: chrome.runtime api. https://developer.chrome.com/extensions/runtime/ Accessed on 2016-08-04.

[19] Google: chrome.storage api. https://developer.chrome.com/extensions/storage/ Accessed on 2016-08-04.

[20] Fawcett, T.: An introduction to roc analysis. Pattern Recogn. Lett. **27**(8) (2006) 861–874