A Quantitative Comparison of Semantic Web Page Segmentation Algorithms

Master's Thesis in Computer Science
by

Robert Kreuzer

Daily Supervisor:
Dr. Jurriaan Hage

Advisor:
Dr. Ad Feelders

June 18, 2013

Created at

Universiteit Utrecht

Abstract

This thesis explores the effectiveness of different semantic Web page segmentation algorithms on modern websites. We compare the BlockFusion, PageSegmenter, VIPS and the novel WebTerrain algorithm, which was developed as part of this thesis, to each other. We introduce a new testing framework that allows to selectively run different algorithms on different datasets and that subsequently automatically compares the generated results to the ground truth. We used it to run each algorithm in eight different configurations where we varied datasets, evaluation metric and the type of the input HTML documents for a total of 32 combinations. We found that all algorithms performed better on random pages on average than on popular pages. The reason for this is most likely the higher complexity of popular pages. Furthermore the results are better when running the algorithms on the HTML obtained from the DOM than on the plain HTML. Of the different algorithms BlockFusion has the lowest F-score on average and WebTerrain the highest. Overall there is still room for improvement as we find the best average F-score to be 0.49.

Drum, so wandle nur wehrlos Fort durchs Leben, und fürchte nichts!
(Friedrich Hölderlin)

Acknowledgments

I would like to thank Stefan, Rob and especially Mustafa for their tireless help in creating the two datasets used in this thesis. I would also like to thank my parents for their continued support during my studies, which seem to have come to an end after all.

Then I would like to thank Ad Feelders for his valuable advice regarding empirical studies and statistics. And most of all I would like to thank Jurriaan Hage for the many discussions we had and his many ideas and suggestions which made this thesis much better in the end.

Finally I would like to thank Ola for putting up with me pursuing my studies abroad for the last three years.

Contents

Chapter 1

Introduction

1.1 Topic

In this thesis we explore the problem of algorithmically segmenting Web pages into so-called *semantic blocks*. The problem can be described as follows: Humans can intuitively infer which parts (or blocks) on a Web page belong together simply by looking at them. It is immediately clear to them what constitutes a menu, an advertisement, an article, a headline etc. This is even true in most cases for languages which the reader does not know, by recognizing visual features such as the positioning of a block, the font size, the styling and the experience the user has with similar pages.

We are now interested in the question whether it is possible to teach computers this ability that humans possess naturally. We therefore look at a number of existing approaches that try to solve this problem in different ways. They can be divided into vision-based ones, that try to imitate the way humans do the segmentation by looking at the rendered document, structure-based ones, which take the underlying structure of Web pages into account, given to them by the document creators, and language-based ones, which only look at linguistic features of Web pages to infer the segmentation. We therefore took one algorithm from each class and compared them systematically on a common dataset using a testing framework we developed for this thesis. We then used the results from the evaluation to create a novel algorithm that combines some features of the algorithms analyzed previously. We present the results of our evaluation and discuss how well computers currently perform on the segmentation problem and what remains to be done.

1.2 Relevance

We posit that a robust solution of the segmentation problem would be highly useful and could be utilized in many different applications. One of the most intriguing prospects is the application of such a solution to the Semantic Web. The hype that surrounded the Semantic Web in the early 2000s, promising to turn the Web into a giant distributed database of all human knowledge, has since died down. Outside of research projects there have been almost no applications of semantic technologies such as RDF[2], SPARQL[25] and OWL[21][14]. The only

technique that has gained some popularity with Web designers is the addition of some semantic tags in HTML5[17] such as `<nav>`, `<article>`, `<footer>`, `<section>` and a few more. While some modern websites make use of these, the vast majority of sites still does not. On those sites a segmentation algorithm could be helpful to enable some basic reasoning and querying capabilities as were imagined in the original vision of the Semantic Web.

Another area where a robust segmentation algorithm could be of great use is the Mobile Web. The number of mobile users is rising rapidly and is poised to overtake the number of desktop users in the next few years. Especially in developing countries many people only have access to mobile devices and more and more are able to access the Internet. The problem one quickly encounters when looking for information on a mobile device is that most websites have not been optimized for them, meaning they will either not render properly or require the user to zoom in and painstakingly hunt for the information she is searching for. This tedious and inefficient access to information is solved on pages which have been optimized for mobile viewing and which are easily readable and navigable by thumb instead of by mouse.

A segmentation algorithm that returns the semantic blocks of a page could thus be used as the basis for a page adaptation algorithm that arranges the blocks vertically and puts the menu at the top of the site, to make it easily navigable. Such a technique could potentially be integrated into a browser and be triggered by the user, if a page does not offer a mobile-optimized version.

More concretely we believe the relevance of our research lies in the establishing of a general testing framework that allows to compare different algorithms on a shared (but exchangeable) dataset using user-defined metrics. This allows for easier exploration of the problem space of the segmentation problem and also makes the results of different algorithms comparable and the effect of parameter changes immediately testable.

1.3 Applications

The applications to the Semantic and the Mobile Web mentioned above seem the most compelling to us, but there are a number of other fields where segmentation algorithms can be used. One is search, where a simple classification of blocks into *noise/no-noise* can help with directed crawling and improving search results in general. A crawler could be instructed to only follow links in what are considered relevant blocks, thus reducing the search space considerably and allowing it to crawl more pages in the same time, while most likely increasing the quality of results.

If a more elaborate ontology is used then various information retrieval and information extraction tasks are possible, like e.g. only getting the main article from a page and discarding everything else or retrieving all images. Getting the main content is useful if there is only limited space for displaying results, like on a mobile device, or when searching for specific information.

Other possible applications are duplicate detection and phishing detection. Here one would either look for duplicates on the same page or across different pages or websites. Looking at blocks instead of whole pages has the advantage that you can make more fine-grained distinctions. As different pages could be quite different in total but still share a number of blocks with each other.

Finally there are caching and archiving. If some blocks are only rarely changed, such as e.g. content blocks as opposed to advertisements, they could be cached to enhance page load times. Similarly if content needs to be archived, one could only extract the relevant parts and archive those, thus reducing the required space considerably.

1.4 Thesis structure

In Chapter 2 we give an overview of the research. We discuss the problem and present the research question. We also introduce the methods we used in our evaluation. The definitions of the basic concepts are given as well as a more in-depth overview of the different approaches to page segmentation.

Chapter 3 is a literature review where we give a short summary of the papers we found the most relevant to our research. We highlight the different approaches and methodologies used to evaluate the results.

In Chapter 4 we focus on datasets where we look at how other authors chose their samples, what was included in those samples and how they marked up the blocks. We then also explain why we had to build our own dataset and how we approached that.

Chapter 5 introduces the testing framework we developed to compare the different segmentation algorithms. We talk about the requirements, the architecture and the implementation of the software. It is shown how new algorithms and metrics can be integrated and how different datasets can be used.

In Chapter 6 we describe the BlockFusion[18], PageSegmenter[30] and VIPS[10] algorithms. We present the different approaches each of these algorithms take to tackle the problem and we talk about the implementation of the ones we implemented ourselves. We also show how they are integrated into the testing framework. Furthermore we describe the new WebTerrain algorithm which was developed as part of this thesis.

Chapter 7 is a summary of the empirical results we got for all four algorithms. These results are discussed in Chapter 8 and we conclude in Chapter 9.

Chapter 2

Research Outline

2.1 Problem discussion

The research on structuring information on Web pages into semantic units goes back at least to 1997, with a paper from Hammer et al. [16]. In that paper the authors describe an extraction tool, where the user can declaratively specify where the data of interest is located and how the extracted data should be turned into objects. This tool was thus designed to work on specific websites where the user knew in advance where the information she was looking for was located. Subsequent authors tried to automate this process, which can be broken into two distinct steps. The first one is simply recognizing which parts, or blocks, of a page belong together, without knowing the meaning (or type if we think in terms of objects) of them. The second step is the labeling of these blocks, i.e. assigning a classification to them.

The first step can be done independently of the second, since additional information such as the structure and the visual rendering of the page are available. This is e.g. done by Cai et al. [10] who first build a content-structure tree out of the recognized blocks and then attempt to label those blocks. For the first step one has to decide whether a flat segmentation, i.e. no sub-blocks, is sufficient or whether a recursive one is wanted. For the latter one has also to decide on a granularity, since in theory you can have any number of tree levels. Most authors seem to have decided on a flat segmentation, though.

The labeling step is more ambiguous since it relies on the ontology chosen by the authors. It depends on the wanted application whether a simple *noise/no-noise* classification or a comprehensive one with many different types is required. This step is typically done using machine-learning techniques which are sometimes augmented using domain-specific knowledge.

We will only be focusing on the first step in this thesis because it lies at the foundation of the segmentation problem. If the quality of the recognized blocks is not sufficient then any subsequent labeling will not make much sense, because blocks will have been cut in half or different blocks will have been joined together.

While extensive work has been done in this field as documented by Yesilada [31], the question remains how well different approaches stand the test of time. The Web has evolved rapidly in the last 15 years and become much more

sophisticated and complex. While in the beginning you had a simple mapping from a URL to a static HTML page, this has become more and more dynamic with the browser now being akin to its own operating system running many different kinds of applications up to full-blown 3D-games[23]. This dynamism poses a challenge for any kind of Web page analysis on which we will focus in the following.

2.2 Research question

Given the problem stated above the research question then becomes:

> *How well do existing Web page segmentation algorithms work on modern Web sites and can they be improved?*

Sub-questions

The research question leads to a number of sub-questions that need to be answered in order to answer the question itself:

1. What are the current best-performing algorithms tackling this problem?
2. What assumptions do those algorithms make and are they still valid?
3. How do they define a "semantic block"?
4. In which scenarios does their approach work well and in which not?
5. How can you compare these different approaches effectively?
6. How can these approaches be improved?

2.3 Methods

We first conducted a literature review to find the most promising looking algorithm from each of the three different approaches described in section 2.5. We consider each one a representative of the class of algorithms which follow that approach and we contrast the theoretical advantages and disadvantages of them. From this analysis we have then developed our own segmentation algorithm which combines some of the features of the others. The main contribution is then an empirical evaluation of these four algorithms on two different datasets which we created for this thesis.

One dataset comprises only popular websites taken from 10 different top-level categories from the Yahoo directory website[1]. The other one comprises randomly selected Web pages from the Internet generated by a random website generator[2]. We used the different datasets to investigate the question whether the segmentation algorithms perform differently on popular and on random pages.

The pages from both datasets were marked up with their semantic blocks by three volunteers to create the ground truth. This was done with a web-based

[1] http://dir.yahoo.com
[2] http://www.whatsmyip.org/random-website-machine/random/

tool which we created for this purpose. They were asked to add two levels of blocks, i.e. top-level blocks and sub-blocks of those blocks and also to assign a classification to each block from a predefined ontology. The classification was ultimately not used in this thesis, but it does increase the usefulness of the datasets and can be used by others or in future work.

One goal was also that the evaluation of the algorithms on a dataset should be completely automatic and not require any additional manual work. For this reason we developed a testing framework described in chapter 5 which simply takes an algorithm and a dataset as inputs, runs the algorithm on the dataset and compares the generated results to the ground truth. The framework can easily be extended with different algorithms and datasets as long as they provide a mapping that adheres to the framework's interfaces.

As metrics for the performance of an algorithm we chose precision, recall and F-score. So we look at how many of the blocks returned by an algorithm are correct, how many of the correct blocks are recalled and at a combination of both measures.

We implemented three of the four algorithms ourselves since reference implementations for all but one were unfortunately not available.

2.4 Definitions

We define a *semantic block* as follows:

> A contiguous HTML fragment which renders as a graphically consistent block and whose content belongs together semantically.

Note that this definition allows for a hierarchy of blocks, i.e. bigger blocks can be made up of a collection of smaller blocks (e.g. the main content on a news website can be made up of a list of articles). In principle there is no limit to the level of nesting of blocks, but in practice it is rarely more than two or three levels.

We will call the number of levels of the block hierarchy the *granularity* of the segmentation. In the following we will use the words *semantic blocks* and *segments* interchangeably. A segmentation with a granularity of one is a *flat segmentation,* since there are no blocks that are subsets of other blocks (all blocks are disjoint).

When referring to the *ground truth,* we mean the set of semantic blocks on a page that have been manually marked up by our assessors. These are considered the true blocks to which the results of the algorithms are compared.

We use the term *Web page* or simply *page* when referring to a single HTML document. The term *website* is used for a collection of Web pages found under the same top-level URL.

The acronym DOM stands for the *Document Object Model*, which is a tree-like representation of XML and HTML documents as objects. It allows to query and manipulate the tree via a standardized application programming interface (API).

2.5 Overview of approaches to Web page segmentation

We give a short overview of the three main approaches to Web page segmentation. We explain the principal ideas behind them and what advantages and disadvantages each of them has.

2.5.1 DOM-based approaches

In a DOM-based approach one simply looks at the DOM (i.e. the tree built by parsing the HTML. Note that this does not include the properties added by potential external CSS files; these will only be present in the render tree) for cues how to divide a page. The idea behind this is that the HTML structure should reflect the semantics of the page, but this is of course not always the case. The quality of these approaches thus depends on the quality of the underlying HTML. To do the segmentation they rely on detecting recurring patterns, such as lists and tables, and on heuristics, like e.g. headline tags working as separators and links being part of the surrounding text.

Advantages

- easy to implement, since one only needs to parse the HTML code and not render the tree
- efficient to run because no browser engine is involved, thus suitable for segmenting large numbers of pages
- take the structural information into account

Disadvantages

- based on the assumption that the HTML document reflects the semantics of the content, which is not necessarily true
- there are many different ways to build the HTML document structure while the semantics stay the same
- disregard styling and layout information by design
- do not work with pages built via Javascript (or you have to serialize the DOM in that case first)

2.5.2 Visual approaches

Visual approaches work the most similar to how a human segments a page, i.e. they operate on the rendered page itself as seen in a browser. They have thus the most information available but are also computationally the most expensive. They often divide the page into separators, such as lines, white-space and images, and content and build a content-structure out of this information. They can take visual features such as background color, font size and type and location on the page into account.

Advantages

- take the styling and layout information into account
- work similar to how a human performs the task
- can take implicit separators such as white-space and vertical/horizontal lines into account

Disadvantages

- more complex because they require a browser engine to render the page first
- computationally expensive because the page needs to be rendered
- requires external resources such as CSS files and images to work correctly

2.5.3 Text-based approaches

The text-based approaches differ from the other two in that they do not at all take the tree structure of the HTML into account. They only look at the text content and analyze certain textual features like e.g. the text-density or the link-density of parts of a page. These techniques are grounded in results from quantitative linguistics which indicate that statistically text blocks with similar features are likely to belong together and can thus be fused together. The optimal similarity threshold depends on the wanted granularity and needs to be determined experimentally.

Advantages

- fast, since the DOM does not need to be built
- easier to implement, since no DOM access is necessary
- comparative performance to other approaches

Disadvantages

- do not work with pages built via Javascript (or you have to serialize the DOM in that case first)
- do not take structural and visual clues into account
- recursive application on sub-blocks requires (arbitrary) changes to the text-density threshold

Chapter 3

Literature review

In this chapter we take a look at the previous work in the field. We give a short summary of the papers we consider the most relevant to our research and point out the many different approaches that have been tried to solve the Web page segmentation problem. We also focus on whether authors did an empirical evaluation of their algorithms, on what kind of dataset this was done and what performance metrics were chosen.

3.1 Related work

In *Semantic Partitioning of Web Pages*[30] the authors develop an algorithm that uses the structural and presentation regularities of Web pages to transform them into hierarchical content structures (i.e. into "semantic blocks"). They then proceed to tag the blocks automatically using an abstract ontology consisting of *Concepts, Attributes, Values* and *None* (their approach is structural and requires no learning). Their main idea behind the labeling is that content is already organized hierarchically in HTML documents, which can be used to group things. They test their results experimentally against the TAP knowledge base[15] (which seems to be not available anymore at this time unfortunately) and with a home-made dataset consisting of CS department websites.

Their approach does not have domain specific engineering requirements and they do not require Web pages to share a similar presentation template. Technically their approach can be split into four parts: They first partition Web pages into flat segments (i.e. disjoint blocks without a hierarchy) by traversing the DOM[22] and segmenting content at the leaf nodes. They define a block as a "contiguous set of leaf nodes in a Web page, where the presentation of the content is uniform."

The second step is inference of the group hierarchy. They use an interesting solution here, where they first transform each segment into a sequence of path identifiers (using the root-to-leaf path of the tree nodes), which are then interpreted as a regular expression, in which each Kleene star represents a separate group. They then proceed to determine labels of groups, where they use the leaf tag right above a certain group if there is only one or a learning-based approach if there is more than one.

The final step is the meta-tagging of groups, where a *Concept* defines the

general context of a group, an *Attribute* corresponds to a group label and a *Value* is an instance of an attribute (i.e. a group member).

In *Identifying Primary Content from Web Pages and its Application to Web Search Ranking*[29] the authors also first segment a Web page into semantically coherent blocks. They then rate the individual blocks by learning a statistical predictor of segment content quality and use those ratings to improve search results.

The segmentation is done using a widening approach, that first considers each DOM node to be its own segment and subsequently joins it with neighboring nodes according to certain heuristic rules. The classification of blocks relies on a machine learned approach where the feature space contains both visual and content properties. The content is simply classified into content segments or noise segments.

In *Recognition of Common Areas in a Web Page Using Visual Information: a possible application in a page classification[20]* the authors present an approach to extracting semantic blocks from Web pages which is based on heuristics that take visual information into account. To get access to information such as the pixel sizes of elements on a virtual screen and their position on the page they build their own basic browser engine. They make some simplifications such as not taking style sheets into account and not calculating rendering information for every node in the HTML tree.

They then define a number of heuristics on the render tree and on the assumption that the areas of interest on a page are header, footer, left menu, right menu and center of the page. They partition each page into areas corresponding to this ontology assuming that each item is to be found at a certain location (e.g. header on top etc.). One issue with this is that these assumptions do not hold true any longer in many cases where Web pages are closer to desktop applications (therefore "Web applications").

The authors test their algorithm experimentally by first building a dataset where they manually label areas on 515 different pages, then run their algorithm on the dataset and subsequently compare the manual and algorithmic results. Their overall accuracy in recognizing targeted areas is 73%.

In *A graph-theoretic approach to webpage segmentation[12]* the authors first turn the DOM tree of an HTML document into a graph where every DOM node is also a node in the graph and every node has an edge to every other node in the graph. Each edge is then assigned a weight that resembles the cost of putting these two nodes into the same segment. The weights are learned from a dataset regarded as the ground truth by looking at predefined visual- and context-based features. Finally they group the nodes into segments by using either a correlation clustering algorithm or an algorithm based on energy-minimizing cuts, where the latter is performing considerably better in their empirical evaluation.

Their evaluation is based on manually labeled data (1088 segments on 105 different Web pages). They also test their algorithm by using it for duplicate detection, where they achieve a considerable improvement over using the full text of a page for that purpose.

For defining features to determine the likelihood of two nodes belonging to the same segment they give three contexts in which DOM nodes can be viewed:

First each node can be looked at as a sub-tree containing a set of leaf nodes with certain stylistic and semantic properties. Second the visual dimensions and location of a node give cues about its semantics and relationship to other nodes. Third the particular DOM structure used by the content creators also implies semantic relationships.

In *A densitometric approach to web page segmentation*[18] the authors use an approach that makes use of the notion of text-density as their main heuristic. Instead of analyzing the DOM tree, like many other authors, they instead focus on discovering patterns in the displayed text itself. Their key observation is that the density of tokens in a text fragment is a valuable cue for deciding where to separate the fragment.

In detail, they look at the HTML document itself and first divide it into a sequence of atomic blocks (text that does not contain HTML tags) separated by what they call gaps (the HTML tags between text blocks). They then discern so-called separating gaps from non-separating ones by analyzing the properties of the two text blocks adjacent to the gap. The deciding property is the text flow which they characterize by the density of the text in a block (defined by the number of tokens in a block divided by the number of lines in that block). If the density between two blocks does not differ significantly (less than a predefined threshold value) they will then be fused together into a new block by the proposed `BlockFusion` algorithm. This is repeated recursively until all further fusions would be above the threshold.

They evaluate their approach experimentally using a dataset consisting of 111 pages. They achieve a better precision and recall than *[12]*.

In *Extracting content structure for web pages based on visual representation[9]* the authors use an approach based on the visual representation of a web page that is independent of the underlying HTML representation of the page. They make the observation that content on web pages is generally structured in a way that makes it easy to read, i.e. where semantically related content is grouped together and the page is divided into such blocks.

Instead of looking at the DOM tree representation of a web page, like many other approaches, they develop a recursive vision-based content structure where they split every page into a set of sub-pages (visual blocks of a page), a set of separators and a function that describes the relationship between each pair of blocks of a page in terms of their shared separators. They deduce this content structure using the VIPS algorithm[10] which goes top-down through the DOM and takes both the DOM structure and the visual information (position, color, font size) into account. Specifically they decide for each node whether it represents a visual block (i.e. the sub-tree hanging on that node) by using heuristics, such as does a sub-tree contain separators like the <hr> tag (if yes, subdivide further) or does the background color of the children of a node differ (if yes, subdivide).

They detect the set of separators visually by splitting the page around the visual blocks so that no separator intersects with a block. Subsequently they assign weights to the separators, again according to certain predefined heuristic rules. From the visual blocks and the separators they can then assemble the vision-based content structure of the page.

The heuristic rules in this paper are determined by common sense. An interesting addition to this work could be to instead determine the heuristic rules using machine learning techniques.

They test their algorithm experimentally by sampling 140 pages from different categories of the Yahoo directory and running their algorithm on it and then manually assessing whether the segmentation was "Perfect", "Satisfactory" or "Failed".

In *Learning block importance models for web pages[27]* the authors of the VIPS algorithm build upon their previous work and they develop an algorithm to rate Web page blocks according to their importance. To ensure that people on average have a consistent notion of the importance of different blocks they first conducted a user study, which confirmed their assumption.

They then develop a learning-based algorithm that takes the position and size of a block as well as some features of the content itself into account (e.g. link density) which are then fed into a Support Vector Machine. According to their experimental results they achieve a quite high accuracy in their predictions.

In *Browsing on Small Screens: Recasting Web-Page Segmentation into an Efficient Machine Learning Framework*[8] the author uses a different approach than most others, because he focuses on the application of optimizing existing Web pages for mobile phones. He first divides a Web page into a 3x3-grid, where the user has to choose one grid he would like to see better by clicking on it. The selected part can then either be zoomed further into or viewed as rendered HTML or transcoded into a view optimized for the specific device.

Their page segmentation algorithm is based on clues from the DOM combined with a number of computer vision algorithms. Specifically they use an entropy measurement to construct a decision tree that determines how to segment the page. They first recursively segment the page by cutting it based on entropy (trying to reduce the entropy of split parts). They combine this with a few heuristics e.g. favoring cuts that result in more equally sized parts or preferably cutting in nodes which are higher up in the DOM tree.

They test their approach on a number of popular websites where they achieve good results in most cases (they rarely cut through coherent texts).

One artificial limitation of their approach is that it is designed to divide the page into at most 9 segments (because they assumed that people can then choose one part by clicking the respective number on their mobile phones), but it seems possible to adapt it to other grid sizes.

In *Vision Based Page Segmentation: Extended and Improved Algorithm[7]* the authors improve upon the popular VIPS algorithm. They focus on improving the first part of VIPS, the visual block extraction (part 2 and 3 of VIPS are visual block separation and content structure construction respectively). They observe that the original algorithm now has certain deficiencies due to its age (it is from 2003) and the evolving nature of the Web. They address these issues by dividing all HTML tags (including the ones introduced by HTML 5) not into three classes but into nine instead and define new separation rules for these classes based on visual cues and tag properties of the nodes.

Unfortunately they do not give an empirical evaluation of their updated algorithm.

There is also a review (*Web Page Segmentation: A Review*[31]) about all the different approaches to page segmentation attempted so far. The author systematically goes through the literature and answers the five W's (Who, What, Where, When and Why). They look at about 80 papers that are in one way or another related to page segmentation.

As applications for segmentation they list mobile web, voice web, web page phishing detection, duplicate deletion, information retrieval, image retrieval, information extraction, user interest detection, visual quality evaluation, web page clustering, caching, archiving, semantic annotation and web accessibility.

They also look at the work that tries to figure out the function and the importance of segments (thus that tries to label blocks). The approaches here include ones based on heuristics, rules, statistics, machine learning and link analysis where sometimes ontologies are used (either predefined or inferred) for mappings.

The approaches for the segmentation itself can be broadly split into bottom-up vs top-down algorithms. Other differentiators are whether algorithms just look at the DOM or at the visual representation of the page or both. Some algorithms attempt to recognize blocks by first looking for separators such as thin lines or white space. Many algorithms are based on heuristics where the authors make assumptions about the "general layout" of a page. There is also one paper where the segmentation algorithm is based on a picture snapshot of the page[11].

The review also gives an overview of the assumptions and limitations of the different algorithms (unfortunately not in a table though).

In the evaluation of the different approaches there is also a wide variance: Many authors use precision and recall as metrics of effectiveness, others use success rate or accuracy and some also focus on execution time or output size.

One thing that the review also shows is that there seems to be no easy way to compare the different approaches to each other. There seems to be no standardized test for the effectiveness of different page segmentation algorithms. Each paper seems to use its own datasets and test procedures which often have different parameters and goals. It is thus no easy task to decide upon one particular algorithm for a practical purpose at hand.

Chapter 4

Datasets

Since Web page segmentation is a fairly well-studied problem, many authors have done an empirical evaluation of their algorithms. The datasets and methods used for this evaluation vary widely. There appears to be no standard dataset for this problem, instead every author seems to create their own dataset by first randomly sampling Web pages (sometimes taken from a directory site such as http://dmoz.org) and then manually marking up the semantic blocks and often also labeling the blocks according to a predefined ontology.

It seems important to point out that marking up a semantic block and subsequently labeling it are two distinct steps. Marking up a semantic block can be thought of as taking a picture snapshot of a Web page and then drawing a rectangle around all the areas on the page that belong together semantically.

Labeling is the subsequent step where a label, typically describing the function of a block, is applied to each one. The labels used depend on the particular ontology chosen by the authors. These range from a simple noise/no-noise classification to more involved ones containing e.g. header, footer, right- and left-menu, advertisements, content, comments etc.

While the first step is certainly less ambiguous there still remains the question whether there is an intuitive understanding of what constitutes a semantic block among different persons. [30] indicates that there is indeed one. They report an overlap of 87.5% between the 8 test subjects who they asked to mark up the same Web pages. Although this sample is very small we believe it to be evidence enough since it is usually self-evident what belongs together on a Web page and what does not. Our own anecdotal experience indicates that people do indeed mostly agree upon what constitutes a block, but you do have to be specific about the level of granularity you want (e.g. “the most high-level (i.e. biggest) blocks and their most high-level sub-blocks”), since there can be many levels.

Another thing to take into account is that semantic blocks can typically be nested hierarchically. E.g. on a news site there may be a main content area that itself can be further subdivided into a list of articles where each article then consists of a headline, a picture and text. While in principle this nesting could be arbitrarily deep, we find that in practice a block-hierarchy with two levels is sufficient for applications such as information retrieval, information extraction and mobile page-adaptation. We will therefore restrict ourselves to hierarchies of two levels here.

4.1 Other Datasets

Before building our own dataset we investigated the datasets used by other authors to find out how they chose their sample pages, sample sizes and whether they downloaded only the HTML documents themselves or the referenced external resources as well. Furthermore we wanted to see whether any of these datasets would be suitable for our analysis as well.

We found five other datasets which are shown in table 4.1. The manually labeled ones vary in size from 105 to 515, with the exception of the TAP knowledge base[15] at a size of 9,068 which was a semantically labeled database, which was used as a test-bed for the Semantic Web but is unfortunately not available anymore. The Web pages are sampled completely at random in [12], in [18] they are taken from the Webspam UK-2007 dataset[13] comprising over 100 million pages, which is focused on labeling hosts into spam/nonspam, in *[20]* they first downloaded 16,000 random pages from the directory site www.dmoz.org and randomly chose the sample pages from there. In [30] they make a distinction between template-driven and non-template-driven Web pages (i.e. pages generated by a Web page authoring system and hand-coded ones) which is not made by the others.

The labeling of blocks was sometimes done by the authors and sometimes by volunteers , which is preferable to avoid any biases. It is not always mentioned what granularity of blocks was used (i.e. whether only top-level blocks were marked up or sub-blocks as well), but no one specifically mentioned marking up sub-blocks which leads us to the assumption that no sub-blocks were highlighted. Since none of these datasets are available online or from the authors directly we were unable to confirm this though.

One other notable observation is that all datasets seem to consist only of HTML documents without any of the external resources referenced from the pages. While this is certainly partly due to the datasets being already up to 11 years old, when Web pages on average were still a lot less complex than they are now, we will go into more detail why this is suboptimal for our analysis in 4.2.1.

Paper	Sample taken from	Size	Granularity	HTML only?	Type	Created by	Available
Semantic Partitioning of Web Pages[30]	TAP knowledge base[15]	9, 068	?	yes	template-driven	external	-
	CS department websites	240	?	yes	non-template-driven	8 volunteers	-
Recognition of Common Areas in a Web Page Using Visual Information: a possible application in a page classification[20]	Random sites from dmoz.org	515	1	yes	all	the authors	-
A densitometric approach to web page segmentation[18]	Webspam UK-2007[13]	111	1	yes	all	external	-
A graph-theoretic approach to webpage segmentation[12]	Random Web pages	105	?	yes	all	the authors	-

Table 4.1: Datasets used in the literature

4.2 Building our own dataset

We first tried to find an already existing dataset which we could use for our evaluation, by contacting the authors of previous evaluations, but were ultimately unsuccessful, because all but one dataset were not available any more. The one dataset we could obtain was unfortunately unsuitable for our purposes, since it for one was focused on news websites and only had parts of the pages marked as blocks (the header and article) and second it only consisted of single HTML files and was missing all external resources, such as images, CSS files etc.

The latter is a problem in our case, because all algorithms that depend on a rendered representation of the page will deliver very different results for a page with all external resources and one without. We want to lay out this point in a bit more detail, since it is relevant for building a suitable dataset.

4.2.1 External resources in Web pages

Web pages are not only made up of HTML documents but they can reference a number of external resources that the browser will download in order to

render the page properly. The most common ones are images, videos, CSS files (which describe how a page should be styled) and Javascript files (often adding interactivity and animations to a page) but there are also lesser known ones like font files, JSON or XML files (providing raw data), favicons and vector graphic files.

A browser will first download the HTML document itself, parse it into the DOM tree (i.e. an object representation of the document) and then find all the referenced resources and download those as well. If there are one or more external style sheets they will be parsed as well, and the style rules will be applied to the DOM. Finally if there were Javascript files found they will be interpreted by the Javascript engine built into the browser and they will apply arbitrary transformations to the DOM. Finally a render tree can be built from the DOM which is then painted on the screen.

So it is clear that if you only download the HTML document itself then its rendered representation will be vastly different from the page including all resources. For this reason we decided to build a dataset consisting of HTML documents together with all their external resources (and all links rewritten accordingly so that they point to the right files).

> **Note** Javascript in general poses a challenge for any kind of Web page analysis because it makes the Web ever more dynamic. While in the early days of the Web there was a clear relationship between an HTML document and the rendered page this is not necessarily true anymore. Since a Javascript program can modify the DOM arbitrarily and furthermore load in more data or other Javascript programs from external sources it is possible that the original HTML document and the rendered page have virtually nothing in common. There is even a Web design approach called "Single-page websites" where the entire website is built through DOM manipulation (e.g. when clicking on a link the browser will not actually make a new request and rebuild the DOM but instead only the changed data will be loaded and the current DOM will be manipulated accordingly). While this approach certainly goes against the design ideas behind HTML, in particular that an HTML document should describe the structure and content of a page, it still must be taken into account for any Web page analysis.

4.2.2 Downloading the sample pages

One of our goals for the dataset was that it should contain the complete Web pages including all external resources. This would allow the pages to be rendered properly and furthermore it should also increase the lifetime of the dataset significantly and thus make it potentially useful for others as well. This led us to the question of how you can actually download a Web page completely. Our first attempt was a small script that simply parsed the original HTML document and looked for links to images, CSS files and Javascript files which it would download as well. This approach turned out to be too simplistic though, as there are a lot more ways to reference external resources than we originally were aware of. Some of the difficulties are that e.g. CSS files can reference other CSS files (which again can reference others as well of course and the references can be circular) and these references can also be in the HTML document itself since it is possible to have inline CSS declarations there. Then there are more obscure references

to things like font files and background images, which is a CSS feature. Finally Javascript poses a real challenge since it is a full programming language (unlike HTML and CSS which are not Turing-complete) that can arbitrarily reference other resources. In practice this is solved by simply running the Javascript program. But in theory the question of whether really all resources have been downloaded is uncomputable (for all possible programs) since it would be akin to answering the halting problem[28].

Fortunately there are tools that solve this problem satisfactorily in practice. We have had the best success with the command-line program `wget` using finely tuned parameters[1]. It handles all of the difficulties mentioned above except references from within Javascript programs, and it also rewrites the links so that they all point to the right locations. We found that the downloaded pages rendered identical or nearly identical to the online version in most cases. The few pages that used Javascript references to an extent that they could not be properly rendered offline were excluded from the dataset (18 out of 100 for the random dataset and 30 out of 100 for the popular dataset). Next we will describe how semantic blocks were added to the downloaded pages.

4.2.3 The Dataset-Builder

We developed a small tool, which we will simply call the `Builder`, to facilitate marking up Web pages with semantic blocks (since we could not find an already existing tool that fulfilled our requirements). For the reasons described in section 4.2.2 it was necessary that the `Builder` operates on the DOM, since that is what people see in the end, and it therefore had to be implemented in Javascript and run in the browser.

The `Builder` is run by clicking a bookmarklet which will load and run the Javascript program from our server. It then works by highlighting the DOM node the user is hovering over with the mouse, allowing her to select that block and showing her a menu where she can choose what type of block it is. The possible choices to classify a block were:

High-level-blocks Header, Footer, Content, Sidebar

Sub-level-blocks Logo, Menu, Title, Link-list, Table, Comments, Ad, Image, Video, Article, Searchbar

This block ontology was chosen with the goal of being comprehensive and it was divided into High-level blocks and Sub-level blocks (or level 1 and level 2 blocks) since Web pages can be segmented on different levels of granularity. E.g. a content-block can have a title, an image and an article as sub-blocks. While in principle there is no upper limit to how many levels of granularity you can have on a page, we found two levels to be sufficient in the majority of cases and have thus restricted ourselves to that.

When selecting a block (which maps to a node in the underlying DOM tree) it is possible for the user to expand the selection (i.e. move up one or more levels in the tree) and also to group together multiple siblings on the same level in the tree. The latter is important because there is not always a one-to-one mapping

[1]The magic incantation is: wget -U user_agent -E -H -k -K -p -x -P folder_name -e robots=off the_url

from DOM nodes to semantic blocks (although in most of the cases there is, as we found).

Technically the `Builder` adds a new `data-block` attribute to a selected DOM node whose value is either 1 or 2, depending on the level of the block. It also adds a new `data-block-type` attribute whose value is the chosen type of block like e.g. "Content". If multiple nodes were chosen as a block they will be wrapped in an additional `div` node with the new attributes added. The user can thus use the `Builder` to mark up all blocks on a page and when she is satisfied with the result send it off to the server.

The server can then save the version of the page with the added blocks and the one without. This client-server architecture was originally intended to allow multiple people to work on the dataset at the same time and not having to install anything on their computers (save the bookmarklet). The builder would then send change-sets, consisting of Xpaths mapping to blocks, to the server which would download the respective page and apply the blocks to it. We had to abandon this approach though as it turned out that the original HTML document and the DOM tree were in many cases so different that the Xpaths would not map onto the tree any more.

Instead we employed a more robust solution, requiring client and server to be on the same machine, where we would first make the full page available offline using `wget`, then open that page in a browser, load in the `Builder`, and add all the blocks and finally serialize the changed DOM to disk again. This proved to be a solid approach to marking up semantic blocks on Web pages. If one subsequently wants to get out all the blocks of a page one can do so using a simple Xpath query[2].

4.2.4 Selecting the sample pages

We built two different datasets, one containing only popular pages and one containing randomly selected pages. This was done to see if the algorithms performed differently on random and on popular pages on average. For the popular dataset we took the top 10 pages from the 10 top-level categories from the directory site http://dir.yahoo.com/. The chosen categories were Arts & Humanities, Business & Economy, Computer & Internet, Entertainment, Government, Health, News & Media, Science and Social Science. We believe this gives us a representative sample of popular websites, although not of websites in general. We manually checked all websites whether they still rendered properly after having been downloaded and removed the ones that were broken, which left us with a total of 70 popular pages in the dataset.

For the random websites we made use of the web service from http://www.whatsmyip.org/random-website-machine/ to generate 100 links, which we then downloaded. The service boasts over 4 million pages in its database and the only filtering is done for adult content, which makes it sufficiently representative for the Internet as a whole. After removing pages that did not render properly offline we ended up with a random dataset consisting of 82 pages.

The marking up of the semantic blocks on these pages was done by three volunteers. They were instructed to first mark up all the level-1 block they could

[2]Xpath query to get all blocks: '//*[@data-block]'

find and subsequently all the level-2 blocks.

Chapter 5

Testing Framework

We give an overview of the testing framework which we developed to enable us to compare different segmentation algorithms on different datasets. The requirements of the system are described as well as its architecture and implementation.

5.1 Overview

The main idea behind the testing framework is that it allows the user to easily run different page segmentation algorithms on a number of different datasets in one unified setting. Conceptually the framework takes a segmentation algorithm and a dataset as input and produces statistical results as output. The statistics are computed by running the given algorithm on each item of the dataset and comparing the resulting segmentation to the ground truth given in the dataset. The concrete metrics used are Precision, Recall and F-Score of the segmentation.

One design goal of the framework was that it should be flexible enough to handle algorithms implemented by others in potentially other programming languages. Also it should be possible to run these algorithms on varying datasets with different structures and features. We therefore tried to define a minimal interface which would still allow all required operations. In essence, all a segmentation algorithm should need as its input is the HTML of the page (and the external resources referenced from the HTML) it is supposed to analyze. It will then return the recognized blocks in some form which will then have to be normalized to a common structure that can be easily compared to the reference dataset. For the framework we defined the expected outcome as a HTML document which contains the blocks marked up with the additional CSS attributes `data-block` and `data-block-type` whose value is the block level (1 or 2) and the block type respectively.

It is thus necessary to provide a mapping function for each algorithm, that takes the output of the algorithm and maps it to the expected format. Furthermore it can be necessary to provide a small driver function if an algorithm is written in a different language or is a Web service for example. The driver takes the original HTML as input and needs to interface with the algorithm and return the segmentation result.

The result, together with the ground truth, is then fed to the statistics component which calculates the evaluation result, which is then presented to

the user.

5.2 Architecture

The testing framework uses a pipeline as its main design pattern as pictured in Figure 5.2. As inputs it takes the algorithm and the dataset specified by the user and it generates a statistical evaluation of the performance of the algorithm on that dataset as output. The pipeline has four distinct components which have clearly defined interfaces (Figure 5.1).

The first component is the *DatasetGenerator* which is simply a function that knows how to retrieve the original HTML documents and the HTML documents with the manually highlighted blocks (the ground truth). In our dataset this information is e.g. provided by a *mapping.txt* file which simply provides a *url : filepath* mapping. If such a file does not exist (and cannot be generated) for another dataset the user would need to provide her own *DatasetGenerator,* since the layout is not known in advance.

The original HTML document is then fed to the *AlgorithmDriver,* which is a small function unique to each algorithm, that knows how to apply the specified algorithm to the given document. This function needs to be specific to each algorithm since algorithms can be implemented as libraries, executables or Web services, which is unknown in advance. The driver can then interface with the algorithm by some means like e.g. a sub-process or an HTTP request and return the extracted blocks.

Since there again is no unified format for semantic blocks and different algorithms return different formats it is necessary to normalize the data representation of the blocks. The *BlockMapper* component takes care of this. It takes the raw blocks, which can for example be only the textual content that has been extracted or fragments of the HTML, and maps them back onto the original HTML document to produce our standard format. For this standard we decided to use the original HTML where the semantic blocks have been marked up with the two additional attributes `data-block` and `data-block-type`. These attributes are either added to already existing elements in the document or if multiple elements need to be grouped together we wrap them in an additional `div` element and add the attributes there. Attributes with a "`data-`" prefix have the advantage of being valid HTML 5 and were added as a means to attach private data to elements, which does not affect the layout or presentation. Furthermore storing the algorithm results as HTML has the advantage that you can still render the page and see the highlighted blocks (with an appropriate CSS rule added) and they are also easy to query via Xpath or CSS selectors. This component is again needed for each individual algorithm as the formats of the returned semantic blocks differ widely.

Finally there is the *Evaluator* component that takes the normalized Block HTML and the HTML ground truth as inputs and does the statistical evaluation of the algorithm results. It calculates Precision, Recall and F-score by getting the blocks from both documents and checking which match. It also returns the number of blocks retrieved by an algorithm, the number of correctly found blocks (the hits) and the total number of relevant blocks on a page. The equality of blocks is tested with two different metrics: an exact match metric and a fuzzy match metric. For both metrics the blocks are first serialized to text-only (i.e.

$$
\begin{aligned}
&\mathit{DatasetGenerator} &&:: \mathit{Dataset} \rightarrow \mathit{Iterator}\ (\mathit{HTML\ Pages}, \mathit{HTML\ Ground\ truth}) \\
&\mathit{AlgorithmDriver} &&:: \mathit{HTML\ Page} \rightarrow \mathit{Algorithm} \rightarrow \mathit{Blocks} \\
&\mathit{BlockMapper} &&:: \mathit{HTML\ Page} \rightarrow \mathit{Blocks} \rightarrow \mathit{Block\ HTML} \\
&\mathit{Evaluator} &&:: \mathit{HTML\ Ground\ truth} \rightarrow \mathit{Block\ HTML} \rightarrow \mathit{Statistical\ Results}
\end{aligned}
$$

Figure 5.1: The pipeline components and their interfaces

all tags are ignored) and white-space and newline characters are removed as they just add noise. The exact match metric then does a string equality check to find matching blocks (the intersection of the set of found blocks and the ground truth is taken). The fuzzy match metric does a fuzzy string comparison using the SequenceMatcher algorithm in the Python difflib library[1]. We consider strings with a matching ratio > 0.8 as equal for the fuzzy match metric. We believe that the results using the fuzzy match metric are more valuable for most applications as the quality of blocks will still be sufficient for them.

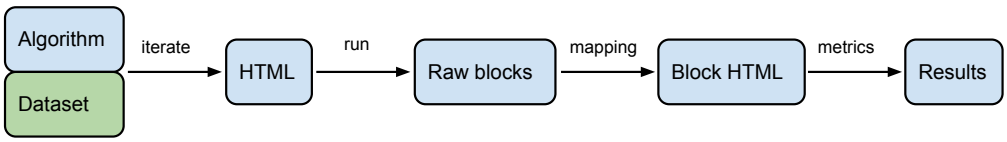

Figure 5.2: The data pipeline

5.3 Implementation

The implementation of the testing framework was done in the Python programming language. Mostly due to the personal preference of the author but one additional reason was that we knew that there is a Python library[6] providing complete DOM bindings, thus making Python a full peer of Javascript, which was a necessary requirement for implementing segmentation algorithms that need to query the DOM. Also there are robust libraries for parsing[3] (often invalid) HTML documents and querying[4] them via Xpath. Furthermore Python programs, which are (typically) interpreted, can easily be modified and extended, making Python a good choice for prototyping and problem exploration.

The *DatasetGenerator* was also implemented as a Python generator (i.e. a lazily evaluated iterator) that reads in HTML documents from disk and yields them when needed. By default files are assumed to be UTF-8 encoded (in our datasets we took care of this when saving the pages) and only in the case of decoding errors we try to guess the character encoding (using the *chardet* library). We do not do it by default as it turned out to be rather slow in practice.

The specific *AlgorithmDriver* and *BlockMapper* functions for the individual algorithms are described in chapter 6. The *Evaluator* parses both the Block

[1] http://docs.python.org/2/library/difflib.html

HTML and the HTML ground truth into a so-called *ElementTree* AST, which supports Xpath queries and (recursive) serialization of nodes to plain text. This is used to first retrieve the blocks via an Xpath query and subsequently serialize them to plain text. We remove white-space and line-break characters from the strings as they just add noise. For our exact match metric we then take the intersection of the set of ground truth strings and the set of algorithm-generated strings to get perfect matches. The perfect matches are then compared to the retrieved and the relevant results to get Precision and Recall respectively. The F-score is then calculated as the harmonic mean of precision and recall.

The results for each page are written to a CSV file with the columns Filepath, Algorithm, Precision, Recall, F-score, number of retrieved results, number of accurate hits and number of relevant results. The results can then be analyzed and visualized with common spreadsheet software.

Chapter 6

Segmentation Algorithms

Here we give a short overview of the other algorithms which we use in our comparison.

6.1 BlockFusion (Boilerpipe)

The BlockFusion algorithm was introduced by Kohlschütter and Nejdl in 2008 [18]. It is thus a relatively recent algorithm which is distinctive in that it is completely text-based. It does not rely on the DOM tree and can be implemented very efficiently, making it potentially useful for the segmentation of a large number of pages. It was included in the comparison as a representative for text-based algorithms.

6.1.1 Algorithm

The BlockFusion algorithm is grounded in the observation, coming from the field of Quantitative Linguistics, that the so-called *token density* can be a valuable heuristic to segment text documents. The token density of a text can simply be calculated by taking the number of words in the text and dividing it by the number of lines, where a line is capped to 80 characters. A HTML document is then first preprocessed into a list of atomic text blocks, by splitting on so-called separating gaps, which are HTML tags other than the <a> tag. For each atomic block the token density can then be computed.

The authors then employ a merge strategy adapted from a Computer Vision algorithm which merges adjacent pixels, but instead they merge neighboring text blocks. They compare the token density of each atomic block to the density of its successor and if the difference between the two (the slope delta) is below a certain threshold value ϑ_{max} they are merged into a bigger block. This merge strategy is done recursively until no more blocks can be merged. Due to this design this algorithm does not support multiple levels of blocks by default, but a different block granularity can of course be achieved by changing the merge threshold value. A possible extension of this algorithm to support sub-blocks is to introduce a second smaller threshold value ι_{max} and then call the BlockFusion algorithm on each (already merged) block. ι_{max} needs to be smaller than ϑ_{max},

because otherwise the sub-blocks would be merged until they are identical to the main blocks.

Algorithm 6.1 The BlockFusion algorithm

```
# Input: textBlocks <- set of atomic blocks which partition the lines
# Output: Set of Text blocks

thetaMax = 0.38    # Empirically determined slope threshold

def blockFusion(textBlocks):
    if len(textBlocks) < 2:
        return textBlocks

    notDone = True

    # merge blocks until no more can be merged
    while notDone:
        notDone = False
        block1 = textBlocks[0]

        for block2 in textBlocks[1:]:
            if slopeDelta(block1, block2) <= thetaMax:
                block1.mergeWith(block2) # merge block2 into block1
                block2.remove() # remove block2 from textBlocks
                notDone = True
            else:
                block1 = block2
    return textBlocks

# The text density difference between two blocks
def slopeDelta(block1, block2):
    d1 = block1.getTextDensity()
    d2 = block2.getTextDensity()
    return abs(d1 - d2) / max(d1, d2)
```

6.1.2 Implementation

While there is no reference implementation of the BlockFusion algorithm as it is given in [18], there is a related open source library from the same author, called boilerpipe[1], which is described in [19]. We implemented the BlockFusion algorithm (specifically the BF-plain variant as described in [19]) on top of this library, since this allowed us to stay as close to the original implementation as possible because we could reuse many functions needed for the algorithm. For example the function that generates the atomic text blocks, which are later merged, is taken unmodified from the boilerpipe library as well as the block merging function and the function to calculate the text density of a block. We also used a text density threshold value of $\vartheta_{max} = 0.38$, which the authors found to be the optimal value in their experimental evaluation.

The boilerpipe library itself is focused on retrieving the main content, like for example an article, from a Web page by removing all so called "boilerplate", i.e. everything that is not the main content, from Web pages. It provides different extractors (strategies) to accomplish this, based on the specific extraction goal. We could therefore simply extend the library with a new *BlockExtractor* that implements the BlockFusion algorithm and returns the extracted text blocks.

We used an existing (minimally modified) Python wrapper for the library to interface with the algorithm[5]. As the blocks are returned as a list of strings they then still need to be mapped back onto the original HTML document. We therefore walk through the parsed HTML tree and search (fuzzily) for the text fragments. When a block is found we add the block CSS attributes (and a wrapping `div` element, if necessary). The thus marked up page is then passed on to the *Evaluator*.

6.2 PageSegmenter

The PageSegmenter algorithm is an example of a DOM-based segmentation algorithm. It works solely on the structural information embedded in the tree structure of a document.

It was introduced by Vadrevu et al. [30] in 2005.

6.2.1 Algorithm

The main idea behind the PageSegmenter algorithm is that the root-to-leaf paths of leaf nodes can be analyzed for similarities to find nodes which likely belong to the same semantic block. An example for a root-to-leaf path would e.g be: `/html/body/p/ul/li`. If multiple adjacent siblings now share this root-to-leaf path it is a pretty safe assumption that they also semantically belong together, as they are structurally part of the same list.

The authors formalized this notion of similarity of paths of leaf nodes as the *path entropy*:

Definition The path entropy $H_P(N)$ of a node N in the DOM tree can be defined as $H_P(N) = -\sum_{i=1}^{k} p(i) \log p(i)$ where $p(i)$ is the probability of path P_i appearing under the node N and k is the number of root-to-leaf paths under N.

Thus one needs to first build a dictionary D which maps all root-to-leaf paths in the tree to their probability of occurrence. For a given node N the path entropy $H_P(N)$ can then be computed by first getting all of the root-to-leaf paths below that node, looking up their probabilities in D and plugging that into H_P.

The algorithm is then described to go breadth-first through the tree and to check for every subset of nodes whether its path entropy is lower than the median path entropy of all nodes. If this is the case then this node is marked as a new segment, else the algorithm recurses into the children of that subset.

We found the original description as given in Algorithm 6.2 to be unclear in two points unfortunately. We tried to get a clarification from the authors but, alas, could not elicit a response. Point one is the formulation "for each Subset S of Nodes[]", which is problematic in two ways: For one, if literally each subset is meant, then the algorithm becomes non-deterministic as the order in a set is not defined. Second it conceptually makes little sense to go through each subset, as that also includes non-contiguous sets of nodes, which - by definition - cannot be segments.

The second point is related, as it also pertains to the question of which subsets are meant. It is the recursion into the children of S which is unclear,

since literally taking all of the children of the different nodes in S and then running the PageSegmenter on that set could lead to the detection of blocks, consisting of non-contiguous nodes (when their parents are different for example).

Algorithm 6.2 The original *PageSegmenter* algorithm

```
# Input: Nodes[], a node in the DOM tree
# Output: A set of segments

for Each Subset S of Nodes[] do
    H(S) := Average Path Entropy of all nodes in S
    if H(S) <= MedianEntropy then
        Output all the leaf nodes under S as a new segment PS
    else
        PageSegmenter(Children(S))
    end if
end for
```

We therefore took the freedom to interpret the algorithm as we think the authors probably meant it, but could not confirm this with them. We will call our interpretation *PageSegmenter′* as outlined in Algorithm 6.3. Firstly we will refer to a list of nodes, as the siblings in a HTML tree have a clearly defined order. Instead of each subset it seems logical to look at each range of contiguous nodes, so for three nodes the possible ranges would be $[[1,2],[2,3],[1],[2],[3]]$. We exclude the whole range $[1,2,3]$ as that has already been checked for the parent of those nodes. We start with the largest ranges since we want to find the main blocks first and then later recurse into them to find the sub-blocks.

If a block is found (i.e. when the average entropy of the nodes in a range is smaller or equal to the median entropy), it is added to the result list and we recursively call the *PageSegmenter′* on the sibling before and after the block. So if we have n nodes and we recognize a block from nodes i to k then we would recursively call $PageSegmenter'([1..i-1])$ and $PageSegmenter'([k+1..n])$ to make sure that we find all contiguous blocks of nodes.

Algorithm 6.3 Our modified *PageSegmenter′* algorithm

```
# Input: nodes <- A list of nodes
#        median <- the median path entropy of all nodes
# Output: A list of segments

def page_segmenter(nodes, median):
    if not nodes: return []

    result = []
    for subset in ranges(nodes):
        average = sum([node.entropy for node in subset]) / float(len(subset))

        # Check if we have a segment
        if average <= median:
            result.append(subset)
            result.extend(split_recursion(subset, nodes, median))
            return result

        # Recurse into children of single nodes
        elif len(subset) == 1:
            result.extend(page_segmenter(subset[0].childNodes, median))
            result.extend(split_recursion(subset, nodes, median))
            return result

def split_recursion(subset, nodes, median):
    """ We recurse into the left and right siblings of a subset. """
    # Get the positions of the first and last element of the subset in nodes
    first = (i for i, x in enumerate(nodes) if x == subset[0]).next()
    last = (i for i, x in enumerate(nodes) if x == subset[-1]).next()
    result = []

    # Get the elements before and after the subset in nodes
    before = nodes[:first]
    after = nodes[last + 1:]

    # Recurse into them
    result.extend(page_segmenter(before, median))
    result.extend(page_segmenter(after, median))

    return result

def ranges(elements):
    """
    A generator for all possible ranges of a list (excluding the list itself).

    Starts with the longest ranges, so e.g. for the
    list [1,2,3] it would generate:
    [[1,2], [2,3], [1], [2], [3]].
    """
    length = len(elements)
    if length > 0:
        for i in (range(length, 0, -1)):
            for x in range(length - i + 1):
                if length == 1:
                    yield elements[x:x+i]
                elif len(elements[x:x+i]) < length:
                    yield elements[x:x+i]
    else:
        yield []
```

6.2.2 Implementation

First we parse the pages into a DOM-like tree (minidom). It is important to choose a tree type which has explicit text nodes, since they need to be leaves in the tree, because the assumption is that all content lies at the leaves. If a tree type like e.g. lxml.etree is chosen where text is simply an attribute of a generic node type this assumption is violated. The next step is to get the root-to-leaf paths of all leaf nodes and calculate the path weights (i.e. how often one path occurs relative to all paths). This information is then used to calculate the path entropy of all nodes, from which we can then get the median path entropy.

Given this information we can then implement the recursive *PageSegmenter′* algorithm. Since the algorithm returns sets of tree elements comprising blocks it is then easy to map these elements back onto the original HTML tree and mark those sets as blocks by adding the `data-block` attribute to the element encompassing the set.

A final note: In their paper the authors mention that they exclude text nodes from the tree where the text contains *modal verbs* (such as could, should, would...) in order to decrease noise. We are not doing that in our implementation as we are not sure how they implemented this and because this would make the algorithm language-specific, which is something we would like to avoid.

6.2.3 Complexity Analysis

This algorithm turned out to be very slow on large pages, which have a lot of nodes on the same tree level. For this reason we evaluated its worst-case run-time complexity and found it to be $O(n^3)$. We will give a short proof in the following.

Assume we have a tree with $n + 1$ nodes. In the worst case n nodes will be direct children of the root node and every single node will be its own block. In that case we will have to first generate all possible ranges for these n nodes and walk through them from the longest ranges to the shortest ones consisting of just one node. The "ranges" in this case are all well-ordered subsets of the n nodes we are looking at. Thus 1 for size n, 2 for size $n - 1$ etc. giving $\frac{n(n+1)}{2}$ ranges. For each range another n calculations are necessary in the worst-case to calculate the average node entropy of that range. Thus giving us a worst-case complexity of $O(n^3)$.

6.3 VIPS

The VIPS algorithm[10] (Vision-based Page Segmentation) appears to be the most popular Web page segmentation algorithm since many other papers reference it or compare their results to it. As indicated by the name this algorithm is based on the rendered representation of a Web page. So it not only takes the DOM structure into account but rendered properties such as dimensions on the screen, background color etc. as well. The algorithm is from 2003 and thus the oldest in our comparison.

6.3.1 Algorithm

The VIPS algorithm was designed to segment Web pages similarly to how humans do it. It thus analyzes the DOM after all the styling information from CSS rules have been applied and after Javascript files were executed (and potentially modified the tree). It is tightly integrated with a browser rendering engine since it needs to query for information such as the dimensions on screen of a given element. One thus has to decide on a fixed viewport size in advance on which the page should be rendered.

Concretely the algorithm first develops a vision-based content structure, which is independent of the underlying HTML document. This structure is built by splitting a page into a 3-tuple consisting of a set of visual blocks, a set of separators and a function that describes the relationship between each pair of blocks of a page in terms of their shared separators. Separators are for example vertical and horizontal lines, images similar to lines, headers and white-space.

This structure is built by going top-down through the DOM tree and taking both the DOM structure and the visual information (position, color, font size) into account. Specifically they decide for each node whether it represents a visual block (i.e. the sub-tree hanging on that node) or whether it should be subdivided further by using a number of heuristics:

- if a sub-tree contains separators like the <hr> tag, subdivide
- if the background color of the children of a node differ, subdivide
- if most of the children are text nodes, do not divide
- if the size of the children differs substantially, subdivide

They detect the set of separators visually by splitting the page around the visual blocks so that no separator intersects with a block. Subsequently they assign weights to the separators, again according to certain predefined heuristic rules. From the visual blocks and the separators they can then assemble the vision-based content structure of the page.

6.3.2 Implementation

We have not implemented this algorithm ourselves, since we could use an existing implementation from Tomas Popela instead[24] (The original implementation is not available any more). He implemented the VIPS algorithm as part of his Master's thesis in Java using the CSSBox[1] rendering engine. Since our testing framework is written in Python, we wrote a small driver function in Java, which we could then call as a sub-process from within Python. The sub-process runs the VIPS algorithm on the given HTML document and writes its result back on stdout.

The result of the used VIPS implementation is an XML file which contains amongst others the information about the recognized blocks. These blocks are given as plain text, i.e. they are stripped of all tags. We therefore wrote a mapping function which maps the found texts back onto the original HTML document, which is the format we need to be able to automatically apply the different evaluation metrics to the result.

[1] http://cssbox.sourceforge.net/

Algorithm 6.4 The *VIPS* algorithm

```
# Input: node <- Render tree (DOM tree with rendering information)
#        level <- 0
# Output: Vision-based Content Structure

blocks = []

def divideDomtree(node, level):
    if isDividable(node, level):
        for child in node.children():
            divideDomtree(child, level+1)
    else:
        blocks.append(node)

def isDividable(node, level):
    if node.isRoot():
        return True
    elif:
        # Check all the heuristic rules for separators
    else:
        return False
```

One thing to note here is that, since we only have the plain text of the blocks, we need to check both the text content of each single node in the tree as well as the serialized plain text of the entire sub-tree of which the current element is the root. If a text node has been marked as a block we wrap it in an additional `div`-element containing the `data-block` attribute, which marks it as a block. Otherwise we just extend the tag of the found element with the block attribute.

We also do a fuzzy comparison on the strings, to account for slight text anomalies due to white space and line breaks for example. As a performance optimization we walk through the tree only once when mapping the text back, since we know that the texts returned by the algorithm are in document-order (i.e. depth-first pre-order).

6.4 WebTerrain

The WebTerrain algorithm was developed as our own contribution to the segmentation problem. The main idea was to see if we can combine the different approaches from the other algorithms in order to improve upon the end result. We were interested in the question whether the approaches can complement each other in the sense that they work better on different sub-tasks of the segmentation problem. As an example: It seemed likely that a structure-based algorithm would do better detecting blocks which inherently have a lot of structure, like tables and list-based menus, than a text-based one.

Additionally we developed a novel heuristic which inspired the name of the algorithm and is described in more detail in the following.

6.4.1 Algorithm

The Firefox web browser has a little known feature which allows the user to see a 3D-rendered version of any website[2]. The result looks similar to a geographic terrain map. This feature works by assigning an additional depth-value to each visible element on top of the common width- and height-values, which are already used in the normal 2D-representation of the page. The depth-value is simply the tree-level of the element. The `<html>` element, which is at the root of each HTML document, would have a depth of 1 for example, and the `<body>` element, as a child of the `<html>` element, a depth of 2 and so forth.

We experimented with this feature on a number of different sites and made the observation that the elevation profile seemed to map pretty well to what we would consider the semantic blocks of a Web page. We formulated the hypothesis that a distinct elevation is a strong indicator for a semantic block. This heuristic has the interesting property that it combines a plain structural approach with a rendering-based approach into one, since it not only takes the DOM tree into account but also the visibility and dimensions of each element. It is not possible to tell by simply looking at the original HTML document how it will ultimately be rendered. One does not know how much space each child of an element will take up on the screen, or if it will be visible at all. For this, one needs to actually render the page (although it does not need to be painted to the screen, of course) using a layout engine like e.g. WebKit[3].

The algorithm is initialized with the root node of the tree and it then goes top-down through the tree, looking at all the children on each level. If there is one child which covers over 90% of the visible area it is considered dominant and the algorithm recurses into that child directly. For each child a number of heuristics are applied to find all the visible elements.

Visible in this case excludes children which are literally invisible, but also separator tags, such as `<hr>`, and most importantly a merge step is executed merging headers with subsequent text elements. The merge step first checks heuristically whether an element is a header, based on either its tag or its computed font size, and if yes, merges it (i.e. wraps the elements in an additional `<div>`) with all subsequent elements up to the next separator. Separators are again heuristically determined and include headers, `<hr>` elements, multiple `<br>` elements and images that have the dimensions of a horizontal line.

If after applying these heuristics more than one child is left, we consider them to be semantic blocks, otherwise we recurse into the single child and repeat these steps (unless we have a leaf node, which is by definition a block, if we actually reach it). If the high-level blocks have been found we call the WebTerrain algorithm again on each of those blocks to find all the sub-blocks. If finer levels of granularity are wanted it is of course possible to recurse more times.

The found blocks are then returned and can then be mapped back onto the input HTML.

[2] To activate 3D View, right-click on the page and click on "Inspect Element", then click on the little cube icon in the lower-right corner of the Inspector pane

[3] http://www.webkit.org/

6.4.2 Implementation

Since the algorithm is based on the rendered representation of a page we needed a way to get such a rendering and interact with it. In browsers this is done by a layout engine which first parses the HTML into a DOM-like representation, and updates this tree with all the additional information from external resources, such as style rules from CSS files and other DOM manipulations via Javascript, and subsequently builds a render tree from it which is then drawn on the screen. Rendering relies on a given device size, which specifies the size of the canvas. In our case we set the viewport dimensions to $1024 * 768$ pixels.

In order to get programmatic access to a layout engine we employed the Python Webkit DOM Bindings[4](short: `pythonwebkit`) library. This library is a virtually complete implementation of the DOM standard in Python, thus allowing all the same queries and manipulations of the tree that have historically only been available in Javascript running directly in a browser.

One drawback of the bindings library is that it is cumbersome to install as it has a lot of specific dependencies which need to be satisfied and it must be compiled from source, which in our case required us to make some small modifications to build scripts and source files to get it to work. Another point of confusion is that there is also a library called `pywebkitgtk` which also provides basic access to Webkit, but it is not to be confused with `pythonwebkit`, as it does not provide full access to the DOM!

Once installed, a so-called WebView object can be initialized with a url and the viewport dimensions. This will trigger the loading of the page and display it in an interactive graphical window, similar to a browser window (except without any menus). Currently there is unfortunately no way to prevent the library from opening a window. As a workaround we simply let the program close the window again immediately after loading it, alternatively it also seems to be possible to open the window in a virtual frame-buffer if one needs fully headless execution. Another quirk of the library is that it does not support URLs using a `file://` protocol. We worked around that by setting up a local web server to serve the dataset files.

The WebView object supports callbacks, such as the documentLoaded callback, which is triggered when the DOM is fully loaded, i.e. when all external resources have also been downloaded and applied to the DOM. We start the page segmentation after this callback fired to ensure all elements of the page are available. The algorithm is then implemented as outlined in Algorithm 6.5.

One difficulty implementing the algorithm was caused due to the mismatch of the asynchronous callback-based nature of a WebView and the synchronous implementation of the rest of the framework. Concretely there was no way in the library to trigger a callback after the algorithm finished running. We solved this by running the whole WebView in a sub-process which would write the output HTML as a side-effect to stdout. The framework process would simply block while the algorithm was running and only continue its execution after all data was received.

[4] http://www.gnu.org/software/pythonwebkit/

Algorithm 6.5 The *WebTerrain* algorithm

```
# Input: node <- Render tree (DOM tree with rendering information)
# Output: blocks <- list of nodes which are semantic blocks

def webTerrain(node):
    result = []
    blocks = highlightBlocks(node)
    result.extend(blocks)

    for block in blocks:
        subblocks = highlightBlocks(block)
        result.extend(subblocks)

    return result

def highlightBlocks(node):
    # Get the children of body
    real_children = self.getVisibleChildren(node, merge)

    # A leaf is automatically also a block if we reach it
    if len(real_children) == 0:
        self.highlight(node, level)
        return [node]

    # Recurse into single children
    elif len(real_children) == 1:
        return self.highlight_blocks(real_children[0], level, merge)

    # If we have siblings, we make them blocks
    else:
        # Highlight them
        for child in real_children:
            self.highlight(child, level)
        return real_children

def getVisibleChildren(node, merge):
    real_children = []

    for child in node.children():
        height = child.offsetHeight
        width = child.offsetWidth

        # Heuristic: Filter out invisible ones
        if height == 0 and width == 0:
            continue

        # Heuristic: Filter out separator tags
        elif el.tagName.lower() in ['hr']:
            continue

        # Heuristic: merge headers with subsequent texts
        elif self.is_header(el):
            div = self.heuristic_merge_header(el)
            real_children.append(div)

        # Heuristic: If one child covers more than 90% of the area, recurse into it
        elif node.offsetHeight * node.offsetWidth * 0.9 < height * width:
            return [el]

        else:
            real_children.append(el)

    return real_children
```

Chapter 7

Results

In this chapter we present the results of our evaluation of the four different segmentation algorithms described in chapter 6. We tested all algorithms in a number of different configurations using the testing framework presented in Chapter 5. First, we tested them on the two different datasets which we created for this purpose: The randomly selected dataset and the popular dataset. The first one consists of 82 random pages and the second one of 70 popular pages, marked up by our assessors. We chose these two types of datasets to test whether the algorithms perform differently on random and on popular pages on average.

As a second variable we ran the algorithms on both the original HTML, i.e. the HTML document downloaded from the source URL via a single GET request, and the DOM HTML, i.e. the HTML document obtained by waiting for all external resources to load and then serializing the DOM. As there appears to be a trend to build pages dynamically on the client-side using Javascript, we were interested to see whether our results would reflect this. It is also of note that our tool to mark up blocks manually was browser-based and thus operated on the DOM, making the DOM HTML the true basis of our ground truth. We believe this is a more sensible basis than the original HTML, since it is what the user ultimately sees when viewing a page, and it also is what the creator of the page intended as the final result.

Finally we used two metrics to compare the generated results to the ground truth, the exact match metric and the fuzzy match metric. Both of them compare the string contents of the blocks to each other. Each block is serialized to only text with all HTML tags removed and white-space and newlines removed as well. For the exact match metric it then simply checks for string equality. This is of course a very strong criterion, as a minimally different string would be counted as false, while for most applications it would likely be perfectly sufficient. For this reason we also do a fuzzy string comparison using the Python difflib library[1]. Concretely we use the `SequenceMatcher` class to check for a similarity ratio of better than 0.8 between strings. This class implements an algorithm similar to the "gestalt pattern matching" approach in [26].

So all together there are four testing variables: algorithms, datasets, HTML-type and metrics. This yielded 32 test runs in total, the results of which are presented in the 8 tables below (each algorithm is in each table to facilitate direct

[1] http://docs.python.org/2/library/difflib.html

comparisons). For each algorithm we show the average Precision, Recall and F-Score values. Precision is a measure of quality that is defined as the number of relevant results out of all retrieved results. Recall is a measure of quantity that is defined as the number of retrieved results out of all relevant results. The F-Score is a combination of the two, defined as $F = 2 * \frac{P*R}{P+R}$. Additionally we also show the average number of retrieved blocks, valid hits (i.e. the number of relevant results returned by the algorithm) and the total number of relevant results (determined by the ground truth). The latter is interesting as it shows the difference in the average number of retrieved blocks and it also shows differences between the two datasets.

In general, the fuzzy match metric will give better or equal results to the exact match metric and we also expected the DOM HTML to do better on average than the original HTML.

> **Note** When studying the results closely one might notice that in some cases the F-Score for an algorithm is lower than the minimum of its Precision and Recall values. This is not due to a mistake on our part, but is caused by the fact that we are looking at *averages* of all three values. It is indeed possible that for each individual result the invariant $F(P, R) \geqslant \min(P, R)$ holds, but still $avg(F_1..F_n) < \min(avg(P_1..P_n), avg(P_1..P_n))$ as the following example illustrates:
>
> If $P_1 = 1, P_2 = 0.1, R_1 = 0.1, R_2 = 1$ then $F_1(1, 0.1) \approx 0.18$ and $F_2(0.1, 1) \approx 0.18$ with $avg(P_1, P_2) = avg(R_1, R_2) = 0.55$ and $avg(F_1, F_2) \approx 0.18$.

7.1 The random dataset

Here we present the results of running the four different algorithms on the dataset consisting of 82 random pages. On average we have 12.24 relevant blocks on a random page.

7.1.1 Original HTML input

BlockFusion returns on average about twice as many blocks as there are relevant blocks, but recall is still very low (i.e. retrieved and relevant results hardly overlap). PageSegmenter returns about four times as many blocks as there are relevant blocks and manages thus to get the highest recall scores but rather low precision. VIPS returns too few blocks on average and has therefore low recall, but precision is higher (for the fuzzy match metric). Finally WebTerrain is the closest in the number of retrieved results to relevant results. Of the retrieved results of WebTerrain about half were actual blocks using the fuzzy match metric.

Comparing the exact to the fuzzy match metric it can be seen that results are considerably better for the fuzzy match metric.

Exact match metric

Precision and Recall are generally very low. BlockFusion and VIPS recognize hardly anything. Precision is highest for WebTerrain and Recall is highest for PageSegmenter.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.03	0.06	0.04	25.99	0.77	12.24
PageSegmenter	0.11	0.27	0.14	46.96	2.97	12.24
VIPS	0.07	0.06	0.06	7.42	0.91	12.24
WebTerrain	0.25	0.22	0.21	10.9	2.23	12.24

Table 7.1: Random-HTML-Exact

Fuzzy match metric

Precision and Recall are clearly better for the fuzzy match metric with the number of hits roughly doubling. Especially VIPS improves substantially, indicating that a number of its blocks were only slightly off from the ground truth. The best F-Score (0.42, WebTerrain) is still rather low.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.06	0.11	0.07	25.99	1.51	12.24
PageSegmenter	0.19	0.45	0.24	46.96	5.24	12.24
VIPS	0.28	0.16	0.17	7.42	1.99	12.24
WebTerrain	0.48	0.43	0.42	10.9	4.5	12.24

Table 7.2: Random-HTML-Fuzzy

7.1.2 DOM HTML input

We can still see a notable improvement when comparing the exact to the fuzzy match metric, but not quite as dramatic as for the original HTML.

The number of retrieved blocks is generally higher (WebTerrain minimally lower), reflecting the observation that the DOM HTML is typically more complex (as mostly things are added, rather than removed).

Exact match metric

BlockFusion is performing poorly, but better than on the original HTML. PageSegmenter again exhibits low precision and high recall. VIPS has the best precision and lower recall, while WebTerrain does similarly on both, giving it the best F-Score.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.08	0.14	0.09	30.96	1.79	12.24
PageSegmenter	0.11	0.39	0.16	65.04	4.47	12.24
VIPS	0.36	0.21	0.24	9.24	2.73	12.24
WebTerrain	0.34	0.29	0.29	10.58	3.04	12.24

Table 7.3: Random-DOM-Exact

Fuzzy match metric

We see about a 50% improvement compared to the exact match metric. WebTerrain and VIPS have the highest precision, and PageSegmenter and WebTerrain

have the highest recall. Compared to the original HTML we see some improvements as well, especially for the VIPS algorithm. Overall we see the highest scores here out of all benchmarks.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.1	0.17	0.12	30.96	2.35	12.24
PageSegmenter	0.15	0.51	0.2	65.04	6.12	12.24
VIPS	0.51	0.26	0.3	9.24	3.33	12.24
WebTerrain	0.57	0.49	0.49	10.58	5.33	12.24

Table 7.4: Random-DOM-Fuzzy

7.2 The popular dataset

Here we present the results of running the four different algorithms on the dataset consisting of 70 popular pages. On average we have 16.1 relevant blocks on a page. The slight variation in relevant blocks is because we had to exclude a few (no more than four) pages for some of the algorithms, as they would not be handled properly due to issues in their implementation (e.g. a GTK window would simply keep hanging).

Between the original HTML and the DOM HTML one can see that the number of retrieved blocks universally goes up, giving another sign that the DOM HTML generally contains more content. Overall the results are again better for the DOM HTML, questioning the use of the original HTML in page analysis algorithms.

7.2.1 Original HTML input

The pattern seen in the random dataset repeats: results for the fuzzy match metric are about twice better than for the exact match metric. Both BlockFusion and PageSegmenter return decidedly too many blocks on average, but only PageSegmenter can translate this into high recall scores. VIPS and WebTerrain are fairly close to the relevant number of blocks.

Exact match metric

The results are generally poor with WebTerrain having the best precision and PageSegmenter having the best recall.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.03	0.06	0.03	72.85	1.07	16.15
PageSegmenter	0.05	0.24	0.08	124.43	4.05	16.22
VIPS	0.07	0.09	0.07	16.72	1.17	16.11
WebTerrain	0.18	0.17	0.16	13.86	2.19	16.09

Table 7.5: Popular-HTML-Exact

Fuzzy match metric

The results are relatively better than with the exact match metric, but overall still not convincing. Again WebTerrain and PageSegmenter are the best for precision and recall respectively.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.05	0.12	0.06	72.85	2.07	16.15
PageSegmenter	0.09	0.42	0.13	124.43	6.74	16.22
VIPS	0.13	0.15	0.12	16.72	2.23	16.11
WebTerrain	0.37	0.35	0.33	13.86	4.81	16.09

Table 7.6: Popular-HTML-Fuzzy

7.2.2 DOM HTML input

Similar to what we saw in the random dataset the improvement from exact to fuzzy matches is smaller than it was for the original HTML, but still substantial.

Exact match metric

The results are overall better than for original HTML with the biggest gains for VIPS and WebTerrain. WebTerrain has both the highest precision and the highest recall in this test.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.03	0.08	0.04	81.75	1.34	16.15
PageSegmenter	0.05	0.27	0.07	163.71	4.56	16.3
VIPS	0.13	0.14	0.12	19.51	2.25	16.11
WebTerrain	0.27	0.28	0.26	14.75	3.77	16.09

Table 7.7: Popular-DOM-Exact

Fuzzy match metric

The results for the popular dataset are the best again, as in the random dataset, when running on the DOM HTML and using the fuzzy match metric. The results for BlockFusion are again the worst. PageSegmenter has again low precision and high recall. Noticeably different is VIPS, as it does not exhibit a high precision, as it did for the random dataset. Recall is similar, though slightly lower. WebTerrain exhibits the highest precision and recall, but precision is 0.1 points lower and recall 0.03 points lower than for the random dataset.

Algorithm	Precision	Recall	F-Score	Retrieved	Hits	Relevant
BlockFusion	0.04	0.12	0.06	81.75	2.13	16.15
PageSegmenter	0.07	0.41	0.11	164	6.68	16.22
VIPS	0.19	0.21	0.17	19.51	3.29	16.11
WebTerrain	0.47	0.46	0.42	14.74	6.43	16.09

Table 7.8: Popular-DOM-Fuzzy

Chapter 8

Discussion

We discuss our findings by comparing the different testing variables and talking about each algorithm in detail. We point out the differences in the evaluation methodologies used by different authors which make direct comparisons difficult. And we mention some of the challenges we encountered during the creation of this thesis. Finally we speculate about how the vision of the Semantic Web could ultimately be realized.

Contrasting the testing variables

As we ran each algorithm in eight different combinations where we varied datasets, evaluation metrics and the type of the input HTML, we can compare and contrast the differences we found.

Random and popular datasets

We created one dataset consisting of random pages and one consisting of popular pages to see whether the segmentation algorithms perform differently on them. As can be seen from our results all algorithms perform virtually always better on the random pages than on the popular pages. We believe this is due to the increased complexity of popular pages, which can be seen from the fact that they on average had 32% more blocks than a random page. Furthermore we also found that a popular page on average consists of 196.2 files in total (this number includes all the external resources referenced from a page), while a random page only consists of 79.4 files on average.

The number of retrieved blocks are also universally higher for all algorithms on the popular pages. But while the number of blocks in the ground truth was only 32% higher, the numbers for the algorithms increased by (much) more than that: BlockFusion 164.1%, PageSegmenter 152.1%, VIPS 111.1%, WebTerrain 39.3%. It thus seems that the algorithms do not scale well with increasing complexity. This could also partly explain why our results are generally less favorable than what has been found in earlier publications, as they are up to 10 years old, and the Web has become much more complex since then. It also shows the need for new techniques that deal well with this increased complexity.

Exact and fuzzy match metric

We found that the number of recognized blocks improved significantly when using the fuzzy match metric as opposed to the exact match metric, as was to be expected. We believe that the results from the fuzzy match metric are generally more valuable since the quality of blocks will still be sufficient for most applications. Furthermore it can easily be adjusted to find more or less precise matches by adjusting the matching ratio.

Original HTML and DOM HTML

Comparing the original and the DOM HTML we found that the results of the segmentation for the DOM HTML are virtually always better, which is true for all algorithms on both datasets. This is due to the fact that the DOM HTML is what the user ultimately sees in the browser, it is thus the final result of the rendering process. While in the past it might have been sufficient to analyze only the original HTML, this is not true any more. As the Web becomes more dynamic and the use of Javascript to manipulate the page becomes more prevalent, there is not necessarily a link between original and DOM HTML any more. This also implies that one cited advantage of text-based segmentation algorithms, namely that they do not require the DOM to be built and are thus very fast, is not true any longer, as even for these algorithms it is necessary to obtain the final HTML for optimal results.

The four segmentation algorithms

We found that the four algorithms in our comparison exhibited a widely differing performance. All together none of them performed well enough to be universally applicable, as the highest average F-Score was a 0.49 (WebTerrain). Our comments here pertain to the test runs using the fuzzy match metric and the DOM HTML because we consider those the most relevant. But the general conclusions hold for the other testing combinations as well.

BlockFusion This algorithm showed the worst performance on both datasets. Both precision and recall are very low (<0.1 and <0.2 respectively). It also returns too many blocks on average (2.5x too many for the random dataset and 5.1x too many for the popular dataset). We could thus not repeat the results from [18]. We conclude that a solely text-based metric is not sufficient for a good segmentation, but that it can be used to augment other approaches.

PageSegmenter This algorithm exhibits low precision and (relatively) high recall (<0.2 and >0.4 respectively). This is due to the fact that it retrieves by far the most blocks from all algorithms (5.3x too many for the random dataset and 10.1x too many for the popular dataset). The number of false positives is thus very high. It would thus be interesting to see if this algorithm could be optimized to return fewer blocks while retaining the good recall rates.

VIPS This algorithm showed the biggest difference between the random and the popular dataset. Precision was high and recall mediocre for the random dataset (0.51 and 0.26 respectively), while both were low for the popular

dataset (0.19 and 0.21 respectively). It is not clear why there is such a substantial difference. The number of retrieved results is slightly too low for the random dataset, while it is slightly too high for the popular dataset (25% too low and 21% too high respectively). In terms of the F-Score the VIPS algorithm had the second-best result.

WebTerrain This algorithm showed relatively high precision and recall for both datasets (both >0.4). It retrieved slightly too few blocks for both datasets (14% too few for the random dataset and 8% too few for the popular dataset). We thus find that a combination of structural and rendering-based approaches enhances overall results. Furthermore the terrain heuristic seems promising. Future work could therefore likely improve upon these results by using more sophisticated combinations of different approaches and heuristics.

Analysis of Variance

We did an analysis of variance (ANOVA) of our complete results (i.e. not based on the averaged data shown in chapter 7, but on our raw results) to test the impact of the four variables `algorithm`, `html`, `dataset` and `metric` on the F-score. We used the statistical language R for the analysis, since it has built-in functionality for it. The results of the analysis are shown in the following table:

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	Significance
algorithm	3	20.59	6.864	319.416	< 2e-16	***
html	1	1.58	1.581	73.569	< 2e-16	***
dataset	1	2.28	2.280	106.119	< 2e-16	***
metric	1	4.44	4.438	206.536	< 2e-16	***
algorithm:html	3	1.26	0.419	19.505	1.74e-12	***
algorithm:dataset	3	0.24	0.080	3.731	0.0108	*
algorithm:metric	3	2.19	0.730	33.983	< 2e-16	***
html:dataset	1	0.17	0.172	8.006	0.0047	**
html:metric	1	0.05	0.052	2.401	0.1214	
dataset:metric	1	0.10	0.096	4.468	0.0346	*
Residuals	2298	49.38	0.021			

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 8.1: ANOVA summary given by R

The significance codes are to be interpreted like this: A significance code of '***' means that the probability of observing an F-score as high or even higher as the value that we actually observed, assuming that in reality the variable concerned has no effect on the score, is between 0 and 0.001% (thus, very small). A significance code of '*' indicates a probability between 0.01 and 0.05, and so forth. Hence, if we adopt the conventional significance level of $\alpha = 0.05$, for all terms except `html:metric` the null hypothesis of 'no effect' will be rejected.

We can therefore see in table 8.1 that the four random variables `algorithm`, `html`, `dataset` and `metric` are all highly significant. The analysis thus confirms our intuition that these factors are relevant for an analysis of Web page segmentation algorithms.

We also included *interaction terms* in the analysis (all the colon-separated variables, such as `algorithm:html` and `algorithm:dataset`). An interaction term is the product of two variables that can in itself be a significant predictor of the outcome. A high significance for an interaction term means that the two variables interact, i.e. the effect on the outcome (in our case the F-score) for a given variable x is different for different values of a variable y (for a given interaction term $x : y$).

In our case we can see that all variables interact with each other with high probability, except for the term `html:metric`. The term `html:metric` not being significant means that the influence of `metric` on the F-score is typically similar for different values of `html`. We can explain this by looking at the results, where we found that the fuzzy metric always returns a higher F-score than the exact metric, regardless of the type of HTML used.

Differences in methodologies

One reason that makes direct comparisons to what has been published in the original papers[18, 30, 9] difficult is the difference in evaluation methodologies. In the VIPS publication[9] the authors did not create a dataset containing the ground truth first, against which they then compared the results of running VIPS on that dataset. Instead they ran VIPS on their sample pages and then manually graded whether the segmentation of that page was "perfect", "satisfactory" or "failed". This approach is problematic on two levels: First, there is the obvious conflict of interest, since the authors themselves are grading the results of their own algorithm. Second, whether a segmentation is "perfect" or "satisfactory" is rather subjective and can thus not be repeated by others.

In the BlockFusion paper[18] the authors did not use Precision and Recall, but instead they used two cluster correlation metrics, namely *Adjusted Rand Index* and *Normalized Mutual Information* to quantify the accuracy of the segmentation. They created a ground truth first manually, although it is unclear whether this was done by the authors themselves or by volunteers. The comparison of segmentation result to ground truth was automated.

In the PageSegmenter publication[30] the authors do use Precision, Recall and F-Score in their evaluation as we did in our evaluation. But differently to us they did not do this for all blocks in general on a page, but they divided the blocks into three classes first (which they call Concept, Attribute and Value) and applied the metrics to each of these classes. This again prevents a direct comparison as this division into three classes is specific to their algorithm and not applicable to other segmentation algorithms.

Encountered problems

We ran into a number of complications and unnecessary inconveniences during the creation of this thesis which we would like to address briefly. The first surprising discovery was that there is no de-facto "standard" dataset on which everybody bases the evaluation of their segmentation algorithm, as is common in other fields such as spam detection (where there is the WEBSPAM-UK2007[1] dataset). This is surprising as it forces every author to create their own dataset,

[1] http://barcelona.research.yahoo.net/webspam/datasets/uk2007/

which on the one hand is a lot of work, and on the other hand makes the comparison of different algorithms much harder. Another surprise was the use of original HTML without any external resources, as opposed to DOM HTML with external resources, which we believe to be a much more realistic and robust choice for the reasons given in section 4.2.1. We have open-sourced our datasets[2] and hope that they will be of use to other people.

The second problem was that we could not obtain the original implementation of any of the three algorithms in our comparison (we could obtain an implementation of VIPS, but it is not from the original authors). This again leads to duplication of work, as we had to re-implement these algorithms, and it makes the results more fragile as it is impossible to prove that they were implemented exactly according to their specification. This is true as often the descriptions of algorithms (and in particular of heuristics) are not specific enough to not require some interpretation of the concrete wording. We described one particular example of these problems in section 6.2. Especially for publicly funded research it seems like a sensible requirement to release related code as open-source software to allow others to repeat results and improve upon that work.

Lastly a technical point: While people in general have a shared understanding of what constitutes a semantic block on a particular level (i.e. top-level, sub-block, sub-sub-block), there can still be a difference in the granularity that a specific algorithm is targeting. This needs to be taken into account when comparing different segmentation algorithms.

The future of the Semantic Web

At the current point in time it seems rather unlikely that the Web in general will become more semantic through an approach like RDF[2], which is the official W3C[3] recommendation for Semantic Web data models, as it has not gained any real traction with Web designers. More lightweight approaches like the addition of some semantic tags in HTML5 seem to us more likely to become more popular in the long term as they are starting to appear more and more on newly developed sites. But this will take time as older sites will typically not be retro-fitted with newer tags.

In the short- to middle-term we see two promising approaches, as they do not require the involvement of the original content creators:

1. the algorithmic approach, which we explored in this thesis
2. a crowd-sourced approach, where users manually add semantic meta-information, similar to how the ground truth was created for our datasets

As seen in our results, there is still more work needed for the algorithmic approach to reach a universally satisfiable level (which we would put at an average F-Score of 0.8). But we do think it shows enough promise to warrant more research in this area. Especially more directed algorithms, that do not try to recover all semantic blocks on a page, but instead just try to recover the main content for example, show fairly good results[19]. As the Web keeps evolving and becoming

[2] https://github.com/rkrzr/dataset-random, https://github.com/rkrzr/dataset-popular
[3] http://www.w3.org/

more dynamic, these approaches will have to be kept up-to-date and there is still a lot of room for new ideas.

We imagine the crowd-sourced approach as people using web-based tools to select parts of a Web page and assigning a semantic type to it. This information could then be made publicly available and allow other people to confirm or improve upon these classifications, similar to how Wikipedia[4] operates. This approach has the advantages that it is not dependent on the creators of a website, it is distributed and arbitrary sites can be added by anyone and it can take advantage of the structural similarities between pages on the same website (e.g. articles on a news website have all the same structure typically). Disadvantages are that changes to websites would most likely need manual updates to the classifications as well and that there is no universal standard which the classifications would adhere to.

Nevertheless the immediate applicability to tasks like Information Extraction, Information Retrieval and mobile page adaptation make this a viable option as well.

[4] http://en.wikipedia.org

Chapter 9

Conclusion

Going back to the first part of our research question, *how well do existing Web page segmentation algorithms work on modern websites,* we can now conclude that their performance in general has gotten worse over time. While all three older algorithms, BlockFusion, PageSegmenter and VIPS, showed a strong performance in their original publications, this does not hold any more on our dataset using recent Web pages. As our analysis using one dataset consisting of random pages and one consisting of popular pages shows, the main reason for this is the increasing complexity of websites and their ever more dynamic behavior due to the increasing prevalence of DOM manipulations via Javascript.

Regarding the second part of the research question, whether the existing approaches can be improved, we showed that this is indeed possible by introducing the WebTerrain algorithm, which consistently had the highest F-scores in our benchmarks. As a combination of structural- and vision-based heuristics it serves as an example that these approaches can be used orthogonally to improve results.

The systematic exploration and testing of the different algorithms was enabled by the testing framework we developed for this thesis. It allows to exchange datasets and algorithms and is also easily extensible with more page segmentation algorithms. It thus forms a solid basis for future work in this field. Promising-looking directions are more sophisticated combinations of different approaches and more directed segmentation algorithms that e.g. only focus on certain segments on a page or that target only specific domains of websites.

As a last word, page segmentation algorithms seem like one possible option to a more semantic Web, but there still remains a lot of work to be done.

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
CSV	Comma-Separated Values
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IE	Information Extraction
IR	Information Retrieval (e.g. text classification, de-duplication and full-text search)
UNIX	From Unics (UNiplexed Information and Computing Services)
URL	Uniform Resource Locator
VIPS	VIsion-based Page Segmentation

Bibliography

[1] boilerpipe - boilerplate removal and fulltext extraction from HTML pages - google project hosting. http://code.google.com/p/boilerpipe/. URL http://code.google.com/p/boilerpipe/.

[2] {RDF Vocabulary Description Language 1.0: RDF Schema}.

[3] html5lib/html5lib-python · GitHub. https://github.com/html5lib/html5lib-python. URL https://github.com/html5lib/html5lib-python.

[4] lxml/lxml · GitHub. https://github.com/lxml/lxml/. URL https://github.com/lxml/lxml/.

[5] python-boilerpipe. https://github.com/rkrzr/python-boilerpipe, . URL https://github.com/rkrzr/python-boilerpipe.

[6] Python webkit DOM bindings. http://www.gnu.org/software/pythonwebkit/, . URL http://www.gnu.org/software/pythonwebkit/.

[7] Elgin Akpinar and Yeliz Yesilada. Vision based page segmentation: Extended and improved algorithm. http://cng.ncc.metu.edu.tr/content/emine.php, January 2012. URL http://cng.ncc.metu.edu.tr/content/emine.php.

[8] S. Baluja. Browsing on small screens: recasting web-page segmentation into an efficient machine learning framework. In *Proceedings of the 15th international conference on World Wide Web*, page 33–42, 2006. URL http://dl.acm.org/citation.cfm?id=1135777.1135788.

[9] D. Cai, S. Yu, J. R. Wen, and W. Y. Ma. Extracting content structure for web pages based on visual representation. In *Proceedings of the 5th Asia-Pacific web conference on Web technologies and applications*, page 406–417, 2003. URL http://dl.acm.org/citation.cfm?id=1766143.

[10] D. Cai, S. Yu, J. R. Wen, and W. Y. Ma. VIPS: a visionbased page segmentation algorithm. Technical report, Microsoft Technical Report, MSR-TR-2003-79, 2003. URL ftp://ftp.research.microsoft.com/pub/tr/TR-2003-79.pdf.

[11] J. Cao, B. Mao, and J. Luo. A segmentation method for web page analysis using shrinking and dividing. *International Journal of Parallel, Emergent and Distributed Systems*, 25(2):93–104, 2010. URL http://www.tandfonline.com/doi/abs/10.1080/17445760802429585.

[12] D. Chakrabarti, R. Kumar, and K. Punera. A graph-theoretic approach to webpage segmentation. In *Proceeding of the 17th international conference on World Wide Web*, page 377–386, 2008. URL http://dl.acm.org/citation.cfm?id=1367549.

[13] Crawled by the Laboratory of Web Algorithmics, University of Milan, http://law.dsi.unimi.it/. Yahoo! research: "Web spam collections". http://barcelona.research.yahoo.net/webspam/datasets//datasets/uk2007/. URL http://barcelona.research.yahoo.net/webspam/datasets//datasets/uk2007/.

[14] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference. *W3C Recommendation February*, 10, 2004.

[15] R Guha and R McCool. TAP: a semantic web test-bed. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):81–87, December 2003. ISSN 1570-8268. doi: 10.1016/j.websem.2003.07.004. URL http://www.sciencedirect.com/science/article/pii/S1570826803000064.

[16] Joachim Hammer, Hector Garcia-Molina, Junghoo Cho, Rohan Aranha, and Arturo Crespo. Extracting semistructured information from the web. 1997. URL http://ilpubs.stanford.edu:8090/250/.

[17] Ian Hickson and David Hyatt. HTML5: a vocabulary and associated APIs for HTML and XHTML. *W3C Working Draft*, 19, 2010.

[18] C. Kohlschütter and W. Nejdl. A densitometric approach to web page segmentation. In *Proceeding of the 17th ACM conference on Information and knowledge management*, page 1173–1182, 2008. URL http://dl.acm.org/citation.cfm?id=1458237.

[19] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. Boilerplate detection using shallow text features. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, page 441–450, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-889-6. doi: 10.1145/1718487.1718542. URL http://doi.acm.org/10.1145/1718487.1718542.

[20] M. Kovacevic, M. Diligenti, M. Gori, and V. Milutinovic. Recognition of common areas in a web page using visual information: a possible application in a page classification. In *2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings*, pages 250 – 257, 2002. doi: 10.1109/ICDM.2002.1183910.

[21] Deborah L. McGuinness and Frank Van Harmelen. OWL web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004. URL http://cies.hhu.edu.cn/pweb/~zhuoming/teachings/MOD/N4/Readings/5.3-B1.pdf.

[22] G. Nicol, L. Wood, M. Champion, and S. Byrne. Document object model (DOM) level 3 core specification. 2001. URL http:

//www.lashkul.info/books/Computers/Programming%20languages%20And%20Technologies/XML/Specifications/DOM3-Core.pdf.

[23] Tony Parisi. *WebGL: Up and Running*. O'Reilly Media, Incorporated, 2012. URL http://books.google.com/books?hl=en&lr=&id=uYnyaBClb3IC&oi=fnd&pg=PR2&dq=webgl+game&ots=Q9B0kcwq6w&sig=fgNitxj3mLhlzYTOEp73JySQVNM.

[24] Tomas Popela. *Implementation of Algorithm for Visual Web Page Segmentation*. PhD thesis, Brno University of Technology. URL http://www.fit.vutbr.cz/study/DP/DP.php?id=14163&file=t.

[25] Eric Prud'Hommeaux and Andy Seaborne. SPARQL query language for RDF. *W3C recommendation*, 15, 2008.

[26] John W. Ratcliff and David Metzener. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, 13(7):46–72, 1988.

[27] R. Song, H. Liu, J. R. Wen, and W. Y. Ma. Learning block importance models for web pages. In *Proceedings of the 13th international conference on World Wide Web*, page 203–211, 2004. URL http://dl.acm.org/citation.cfm?id=988700.

[28] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936. URL http://classes.soe.ucsc.edu/cmps210/Winter11/Papers/turing-1936.pdf.

[29] S. Vadrevu and E. Velipasaoglu. Identifying primary content from web pages and its application to web search ranking. In *Proceedings of the 20th international conference companion on World wide web*, page 135–136, 2011. URL http://dl.acm.org/citation.cfm?id=1963261.

[30] Srinivas Vadrevu, Fatih Gelgi, and Hasan Davulcu. Semantic partitioning of web pages. In Anne Ngu, Masaru Kitsuregawa, Erich Neuhold, Jen-Yao Chung, and Quan Sheng, editors, *Web Information Systems Engineering – WISE 2005*, volume 3806 of *Lecture Notes in Computer Science*, pages 107–118. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-30017-5. URL http://www.springerlink.com/content/u62r27p2720t1173/abstract/.

[31] Y. Yesilada. Web page segmentation: A review. 2011. URL http://wel-eprints.cs.manchester.ac.uk/148/.