Extracting Web Content for Personalized Presentation

Rodrigo Chamun, Daniele Pinheiro, Diego Jornada
João Batista S. de Oliveira, Isabel Manssour

Pontifícia Universidade Católica do Rio Grande do Sul — PUCRS
Faculdade de Informática — FACIN
Porto Alegre — Brazil
{rodrigo.chamun, daniele.pinheiro, diego.jornada}@acad.pucrs.br
{joao.souza, isabel.manssour}@pucrs.br

ABSTRACT

Printing web pages is usually a thankless task as the result is often a document with many badly-used pages and poor layout. Besides the actual content, superfluous web elements like menus and links are often present and in a printed version they are commonly perceived as an annoyance. Therefore, a solution for obtaining cleaner versions for printing is to detect parts of the page that the reader wants to consume, eliminating unnecessary elements and filtering the "true" content of the web page. In addition, the same solution may be used online to present cleaner versions of web pages, discarding any elements that the user wishes to avoid.

In this paper we present a novel approach to implement such filtering. The method is interactive at first: The user samples items that are to be preserved on the page and thereafter everything that is not similar to the samples is removed from the page. This is achieved by comparing the path of all elements on the DOM representation of the page with the path of the elements sampled by the user and preserving only elements that have a path "similar" to the sample. The introduction of a similarity measure adds an important degree of adaptability to the needs of different users and applications.

This approach is quite general and may be applied to any XML tree that has labeled nodes. We use HTML as a case study and present a Google Chrome extension that implements the approach as well as a user study comparing our results with commercial results.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval—*Information Filtering*; H.3.3 [**Information Systems**]: Information Search and Retrieval—*Search process*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DocEng'14, September 16–19, 2014, Fort Collins, Colorado, USA.
Copyright 2014 ACM 978-1-4503-2949-1/14/09 ...$15.00.
http://dx.doi.org/10.1145/2644866.2644871.

Keywords

Web content extraction; Web content filtering; Levenshtein algorithm

1. INTRODUCTION

The printed version of a web page is usually a disappointing document: quite often it has poor layout and too many pages. The reason is that the HTML language used to format the pages was not designed for printing (or for obtaining printable versions of the content) and web browsers render pages with unrestricted boundaries, making the conversion of digital content into physically constrained paper unnatural. The way many pages are designed worsens the experience: In most cases, main content is surrounded with elements such as menus, advertisements, comment boxes and others and all these elements are also transferred to the printed version. Even reading the online version of a page may be in many cases annoying because these elements grab the attention and use too much real estate on the screen.

One solution for these problems is to identify the main content of the page (the part that most readers probably wish to consume) and extract these items for cleaner presentation. This technique is known as content extraction. To ease the manipulation, a HTML page may be represented as the Document Object Model (DOM [12]), which translates the HTML text into a tree structure. In this structure, every element is represented as a node in the tree, with the control elements usually being internal nodes and content being leaf nodes. The goal of content extraction is to find out efficiently and precisely what is desired content and what is not.

In this work we propose a method to solve the content extraction problem. This solution starts when the user actively samples elements on the page: The DOM tree is traversed and used to identify the nodes that were selected, and every node in the tree that is considered similar to the sample is identified as possibly interesting content. Similarity is measured by comparing the node types, their level on the tree and their paths from the root. The user also provides a threshold which defines how different a node might be from the sample to be still considered similar to it. The core of our approach is the path comparison between nodes, which is a variation of the Levenshtein algorithm for string edit distances [9]. After samples are selected from a given web page, any information about the selection may be stored and used later for automatic filtering of pages from the same web site. Thus, interactive filter construction is made only once.

The approach is general enough to be used in any tree that has labeled nodes. The main contribution is that with a few clicks the user is able to select from the page only the interesting elements, whereas the threshold used for similarity allows for better customization. The filter produced from the samples may be applied to the web page already on the web browser and immediately all items that were not selected are removed, and it may also be stored for later use.

We present a Google Chrome extension supporting the approach and extended the filtering procedure so that users may apply a filter to several pages from the same web site at once, obtaining a customized view of a set of web pages. This may be presented as a new HTML document or as a PDF document for printing.

This paper is structured as follows: In Section 2 we provide an overview of web content extraction in the academic literature and existing commercial solutions. Section 3 explains in deeper detail how the proposed method works and how it relates to previous works. In Section 4 we describe an application that implements the method as a Google Chrome extension. Section 5 presents user impressions when comparing our results with the results of an existing commercial solution. Finally, in Section 6 we present our conclusions and goals for future research.

2. RELATED WORKS

Several approaches have been proposed to tackle the problem of content extraction from HTML pages. Most of these try to find the most interesting content by exploring the DOM representation of the page and assigning some relevance to each node, sometimes using visual cues from the rendered page to help in the task. Other approaches assume a domain context (mainly the news article domain) and explore features unique to this domain.

Cai et. al. [5] presented a hierarchical structure for identifying web content based on the page representation that groups page segments that look alike. Each identified segment of the page is a branch in the DOM hierarchy. Under the same parent there are elements that are thought to be in the same section of the page, thus having a similar semantic context. This branching is done according to visual features on the page such as the font used for text elements (color and size), segments that are placed near each other, background colors and natural page divisors such as the <hr> tag.

A semi-automatic approach is proposed by Line et. al. [10]. It uses visual features to guess the main content of a page and the user is able to modify it at will. The algorithm explores the DOM tree of the page looking for leaf nodes and groups them by similarity. The similarity is measured by element properties such as geometry, position, style and tag types. Groups are clustered into blocks that have their importance measured by a heuristic. The block with the highest importance is returned as the main content of the page. The goal of that work is to start from pages that originally have complex layouts and contain a balanced mixture of text and multimedia content and obtain printed versions without the diversity of elements usually presented on a web page.

The approach proposed by Wang et. al. [14] aims at extracting content from news sites. It takes advantage of the fact that most news pages are based on a static template filled with content. This suggests the creation of a wrapper for a web page (without knowing its original template) based on content and spatial features. It is based on machine learning and the wrapper is learned from a few samples from a single site and extended to extract news content from other sites. This technique consists in finding the best sub-tree of the DOM tree, as presented in [10]. The extracted article has exactly the same visual style as the page and the result may still have some unwanted content.

Reis et. al. [13] proposes an approach that also takes advantage of the templates filled with content that many news pages adopt for their presentation. Web sites are crawled and have their pages gathered and clustered according to the similarity of their tree structures. For each cluster, a template is extracted from elements that do not change among the pages in the cluster. Once the most important content of an input page is desired, it is submitted to a classification process to find which cluster it belongs to, then, its content is retrieved by looking for changes between the page and the template of that cluster.

Another filtering process is proposed in Gupta et. al. [6] to provide a clean version of the web page for screen readers used by visually impaired people. Instead of trying to find the most important content, the algorithm tries to eliminate non-content and assumes that what is left is interesting. The elimination is done by removing elements from the DOM using a series of different filtering techniques and the authors report that the algorithm performs well on pages with large blocks of text, such as news articles.

Also trying to provide a clean printable version of a web page, Luo et. al. [11] approach the problem by grouping text segments that appear to be in the same context (i.e. do not have a line break between them) and identify the set of segments more likely to contain the most relevant information, to make sure only segments that belong to the news story are selected.

There are also commercial solutions developed either as native browser features or as extensions. HP Clipper [7] is a web browser extension to extract content semi-automatically. It tries to find the main content of the page and allows users to fine tune the result by manually removing items. CleanPrint [1] is another extension that works similarly, where users manually select the content they wish to remove. Reading View [3] and Reader [4] are native features for the Internet Explorer and Safari browsers, respectively, that automatically selects the content, but the user cannot add or remove elements and they do not work for every page on the web. Finally, Evernote Clearly [2] works exactly as the Reading View and Reader but as a browser extension.

The above methods have different approaches to the problem of recognizing content: Some are based on template recognition, some on hierarchical structure or item context, others do not disclose their methods as in the case of commercial software. In any case, they seldom ask the most important agent of the whole process, the user has very little say in the selection of items. Even when this is done, as is the case of HP Clipper [7] and CleanPrint [1], this has to be done for each web page and the selection cannot be reused for different pages. In our proposal, the user will be the main agent for selection and his selection will be made only once and produce a filter to be applied over and over.

3. PROPOSED METHOD

Our approach for content extraction works on documents represented as a hierarchical structure. In order to identify the most important content of a web page and extract it for later presentation, we use the Document Object Model (DOM) [12]. This representation translates the string of HTML text to a tree structure where every element is turned into a node in the tree, with control elements usually being internal nodes and content being leaf nodes. One naive approach might suggest that all leaves of a given HTML tree are the content, but the superfluous elements are also leaves in the tree and since elements have no semantic information the goal to solve the problem is to find out what is desired content and what is not.

The first developed approach to select content was quite simple: As HTML nodes are sampled from the screen by the user, the HTML path for each of those nodes is stored and the collection of paths represents a selection (or filter) to be used on that web page. When the filtering process runs on another web page it searches for all nodes with exactly the same paths from the root as the sampled ones. Although straightforward, this approach is very restrictive as even the slightest difference in the HTML paths will force relevant nodes to be rejected.

Clearly, such a strict method is not flexible enough to be used with a language as tolerant as HTML, where slight changes in the structure may not affect the visual presentation of content (and thus preserve its logical connection in the interpretation of users). Therefore some kind of tolerance should be inserted into the path analysis, providing for some measure of "closeness" to the nodes originally selected.

In this second approach the process begins when users perform manual sampling of items to be preserved on the web page and associate to each sampled item a threshold representing an acceptable difference between the sample and other candidate elements. With that information each leaf node in the HTML tree has its path compared to the sample. When changes are found, the amount of difference is calculated and if it is less than or equal to the provided threshold, we assume that the nodes are similar and the node is part of the desired content. Clearly, the same approach may be used with any XML tree and is not limited to HTML.

The amount of difference compared with the threshold value is processed as follows: Before comparing paths, a specific weight is assigned to each level of the HTML tree. These weights are used to assure higher penalties to path differences happening closer to the top of the tree and more lenient to differences closer to the bottom of the tree. We define that the tree has total weight 100 assigned to its longest path and distribute that weight across all levels in a way that levels close to the root weight more than deeper levels. This works as follows: For the root node, we divide the total weight assigned to the tree by a damping coefficient, take this value out of the total weight and divide the new value of the total weight by the damping coefficient on the next level. Then we keep doing this process until we reach all levels of the tree. The remaining of the total weight is equally distributed among all levels of the tree. Algorithm 1 describes this process.

For example, Figure 1a shows a HTML tree with the weights of its levels already calculated. In this example the user has selected the node *IMG21* as the sample node (its path in the tree is shown in Figure 1b) and the nodes to be compared to it are *IMG5* and *IMG18* (with the paths shown in Figures 1c and 1d respectively).

Algorithm 1 Algorithm that calculates the weights for the tree levels.

function WEIGHTS($tree$: HTML tree)
  $w \leftarrow 100$
  $h \leftarrow tree's\ height$
  $list \leftarrow \emptyset$
  for $i \leftarrow 0$ **to** h **do**
    $list_i \leftarrow \frac{w}{DAMPING}$
    $w \leftarrow w - list_i$
  end for
  $rest \leftarrow \frac{w}{h}$
  for $i \leftarrow 0$ **to** h **do**
    $list_i \leftarrow list_i + rest$
  end for
  return $list$
end function

To quantify the amount of difference between the paths we add the weights of the levels where changes happen. The first test happens between the paths to nodes *IMG21* and *IMG5*. As presented in Figure 1a, the first difference between *IMG21* and *IMG5* occurs on the second level where the weight is 2.61 and the final result of this comparison is 11.66.

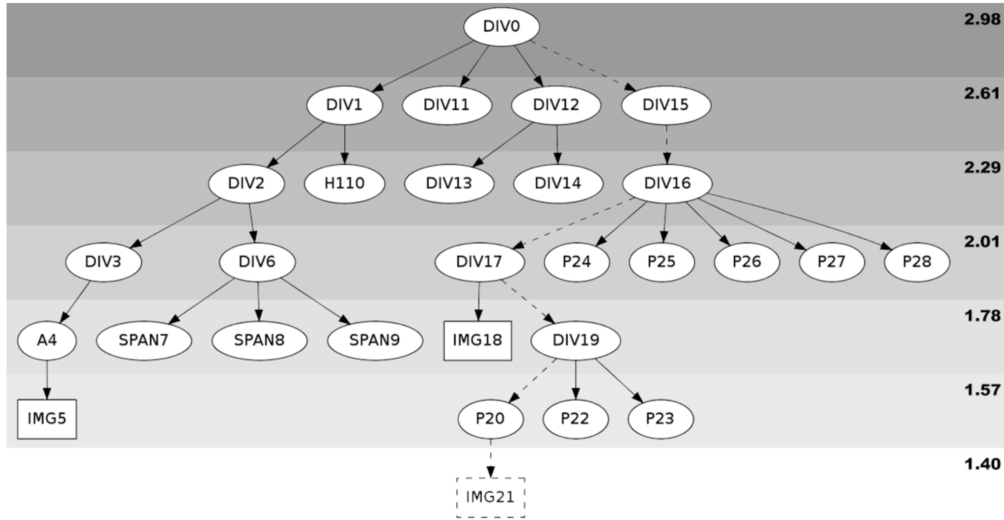
The second test happens between the paths to nodes *IMG21* and *IMG18*. The difference between the paths happens deeper on the tree, on a level of which weight is 1.78 and the final result is significantly smaller, only 4.75.

The threshold provided by the user controls how much one element is allowed to be different from the sample. Following the same example and keeping node *IMG21* as the sampled node, a threshold value of 5 will make node *IMG18* be considered similar whereas *IMG5* will be rejected. A larger value may consider node *IMG5* similar as well at the risk of including more content that is less similar to *IMG21*.

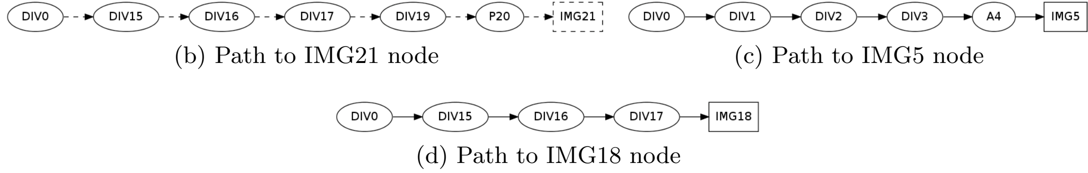
The changes between paths are detected by a variation of the Levenshtein algorithm [9]. This algorithm is originally used to calculate the minimum amount of operations needed to transform a string of characters S into another string S'. The original string operations are character substitution, insertion and deletion and each of these operations adds 1 to the cost of the transformation. Our adaptation to the path comparison problem consists of using this algorithm to find how many operations are needed to transform the path of the current node into the path of the sample, but instead of characters we use node labels and instead of using 1 as the operation cost we use the weight assigned to the level where the operation takes place. This variation of the Levenshtein algorithm is shown in Algorithm 2 below.

3.1 Comparison with other approaches

We do not wish to extract content from web pages autonomously, but rather to provide a tool for users to select samples of what they want and provide some measure of adaptability to select other items that may be similar to the samples. This information can be used later to automatically filter pages that share similar structure with the sampled pages, eliminating the need for the user to sample



(a) Part of a HTML tree: first levels are shown and the weight of each level is on the right



(b) Path to IMG21 node

(c) Path to IMG5 node

(d) Path to IMG18 node

Figure 1: Example of a HTML tree and the weights for each level and paths to three nodes in that tree.

Algorithm 2 Algorithm to calculate the difference between paths

function CALCULATE(a, b: node path, w: list of weights, i, j: path position)
  if $i < 0$ **then**
    return $\sum_{k=0}^{j} w_k$
  end if
  if $j < 0$ **then**
    return $\sum_{k=0}^{i} w_k$
  end if
  $cost \leftarrow w_{max}(i, j)$

  if $a_i == b_j$ **then**
    $cost \leftarrow 0$
  else
    $equal \leftarrow w_{max}(i, j)$
  end if
  return $min(calculate(a, b, w, i-1, j) + cost,$
    $calculate(a, b, w, i, j-1) + cost,$
    $calculate(a, b, w, i-1, j-1) + equal)$
end function

elements again in another page from the same domain. This approach should not be exclusive for news or HTML pages, even though we use them as case study. We also implement a proof of concept application as a Google Chrome extension, to be detailed in Section 4.

As several works [6, 10, 14] our approach explores the DOM structure for extracting content from web pages. On the other hand, the proposed method is not limited to news sites [14, 13] and may be applied automatically after a first sampling made by the user, unlike [3, 4, 2].

Two of the related works have similar approaches to ours: Gupta [6] also proposes a filtering approach in which the DOM structure is navigated and a series of heuristics to remove specific nodes are used. Thus it is expected that content will remain on the page while non-content is removed. This approach also requires user interaction because the heuristics are set by the user. Besides the filtering action, this approach is opposite from ours, since our users must select what they want to keep on the page. It also has a drawback since it requires specific rules for specific types of elements and every undesired type must have specific heuristics associated to it. On the other hand, as our method allows users to select what they want on the page any type of element may be selected with no special rule whatsoever.

The approach by Reis [13] is also similar to ours but depends on crawling to get all pages of a web site, process them in sets of similar pages and store this information. Therefore, the approach works only for a web site that was processed, new websites must have all of its pages crawled to have their content extracted. Our approach allows a very simple filter creation in less than a minute for any page the

users want. Since users know the pages they visit, they know if they are similar, and the filter information is the only thing that needs to be saved. Finally, the creation of a new filter requires only a few clicks and no need to crawl a entire web site.

4. APPLICATION OF THE METHOD

In this section we describe an application that implements our method as an extension to the Google Chrome web browser. The choice of browser was based on its popularity and the availability for several operating systems, but other browsers could have been used as well.

Before content extraction, node sampling is necessary: The user is asked to sample nodes and assign them to several existing node types. Thus, a node assigned to a type called "headline" may be handled differently than a node classified as "text". Similar information could be obtained from the analysis of HTML tags of the nodes, but we chose to avoid that analysis and work with explicit information provided by the user.

For example, to identify all paragraphs of a web page and handle them as normal text later on, the user selects a paragraph element on the page, assigns it to the *text* type, adjusts the threshold and runs the algorithm. Every DOM node similar to the sample is assigned to *text* as well. We provide three default types that categorize most elements on a page: "headline", "text" and "image", but we also provide the option of creating new, personalized types. This personalization allows users to select elements from the page and assign those elements to a new type, as for example date and author. This may be interesting when further processing is to be made on the data obtained from the web page and content may be "tagged" with such types for easier identification.

For node sampling, the user clicks on an element on the page, assigns it to a type and moves a slider to set the threshold as described in Section 3 and the higher it is, less similar a node needs to be from the sample to be selected as similar content. As the slider is changed, all elements that are selected by the algorithm are highlighted on the screen.

Figure 2 shows the extension's menu. The extension works in two modes: The *Clean* and the *Clean All* options. The *Clean* option resets the application and begins to work with samples, so the user may select items to create a new filter for that web page, apply it to the page (and clean it, therefore the name), save the filter or load an existing one. The second option is *Clean All*, where an existing filter can be applied immediately to several links that are selected from the current web page.

Figure 3 illustrates the selection of a sample of text. When the user moves the slider in the extension's menu, the algorithm selects other elements according to the threshold defined by the slider value. Figure 4 depicts the web page after the threshold specification. The process for headline, image and custom elements is the same.

A filter may be applied on the same page used to create it by clicking on the *Apply* button on the extension's interface. When this action is performed, only the elements that match the sampling will remain on the page – thus cleaning the page. Figure 5b presents the results of a filter that keeps only the headline and the paragraphs of the news pages shown on Figure 5a.

Figure 2: Extension's interface

Figure 3: Sampling a piece of text.

The extension also provides the functionality of applying the filter created for a single page to a set of other pages. By clicking on *Clean All*, this may be automated for a number of links on a page. The link selection follows the same approach presented for sampling elements: The user selects a link from the current page as a sample and then moves the slider to select more or less similar links. This is exemplified on Figure 6. Thereafter, clicking at the *Apply* option starts the process: All links that match the sample will be followed and the filter will be applied to them.

The extension receives the results obtained from following the links and sends them to an external tool to create a layout for this content. The tool used in our extension implements the algorithms presented by Oliveira [8] to render

Figure 4: Selected paragraphs of a web page.

(a) Original page.

(b) Filtered page.

Figure 5: A news page before (a) and after (b) content extraction.

pages with columns. The output format is chosen by the user, either as a PDF file or HTML to be rendered by the browser. Figure 7 presents the HTML output of this feature on the browser, with a navigator at the top which paginates the output, a page for each link. Figure 8 shows the PDF output that is downloaded by the user.

Figure 6: Selected links where filters are to be applied.

For a non-news web page, Figure 9a shows the original web page containing a recipe and several other minor elements, including advertising. Figure 9b, on the other hand, shows a much cleaner version of the same web page after filtering, without any ads. It is interesting to notice that in this case we used the existing item types (headline, text, image) and a fourth type was created to select the recipe

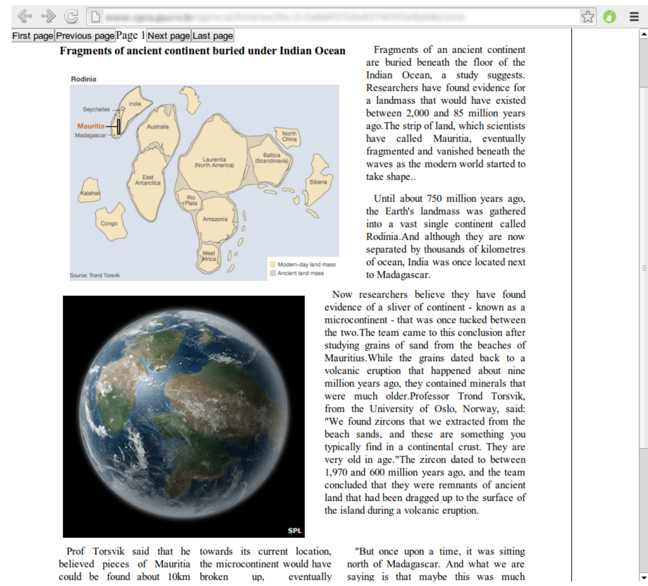


Figure 7: HTML output. Content from `http://www.bbc.com`.

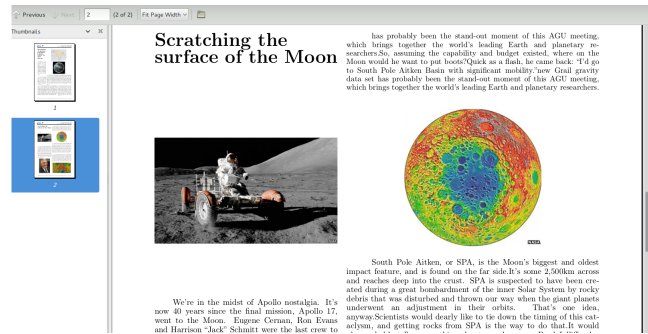


Figure 8: PDF output. Content from `http://www.bbc.com`.

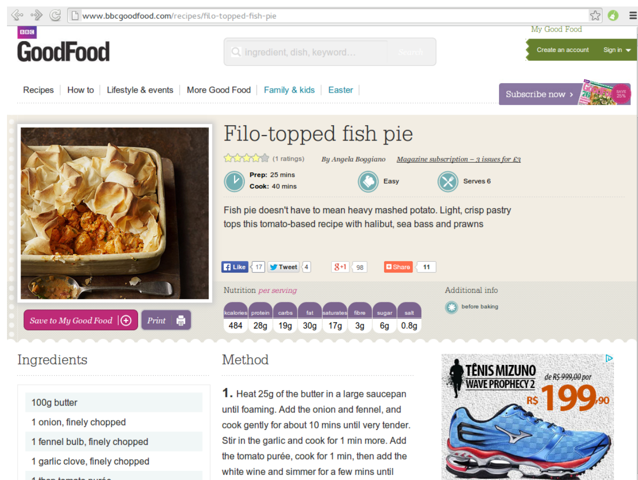
ingredients. Thus, if further processing was needed these ingredients would be readily recognized in the HTML file.

5. USER EVALUATION

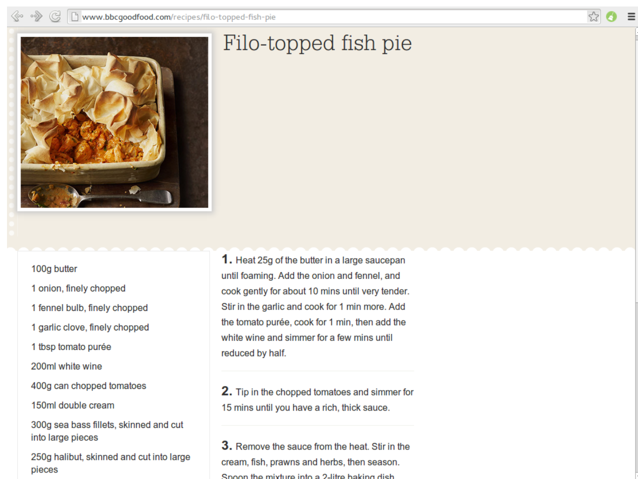
This user evaluation was based on interviews to collect user opinions about the extracted content rather than the prototype usability or the final document layout. Users were asked to compare the output from our prototype with the output of Reading View [3] and point out their preference. Two pages from BBC News[1] and one from G1[2] were selected arbitrarily and used in all interviews. Since the concern was on the tool output only, the users were presented with printed versions of each page and printed versions of the results of each tool.

Eighteen people between ages 19 and 30 were interviewed individually. Their background ranged from computer science to psychology and communication. Each interview was performed as follows: a brief explanation about web content extraction was given to the volunteer and then the printouts of each web page were shown in turn. For each page, the

[1] `http://www.bbc.com/`
[2] `http://g1.com.br/`



(a) Original web page with a recipe, including several extra items and an advertisement.



(b) The content from the page, after filtering.

Figure 9: An example of extraction on a non-news web page.

user was told which printout was the original news page, for the Reading View output the user was told only that a tool generated it automatically, and for the output of our prototype the user was told that another tool generated it with a few clicks. To emphasize that the output could be personalized, two versions of our output (one containing the news headline, text and image and the other with the same content plus the author name and date) were shown as being different outputs of the same tool for the same page. It was never mentioned the name of the application that generated each output and the volunteer was asked to disregard the layouts and evaluate only whether the extracted content interested them in contrast to the original page. After showing all versions of the same news page we asked which one the volunteer preferred and why.

As presented in Figure 10, 9 people preferred the output from our prototype claiming that it is more useful to choose what goes to the output while 7 people preferred Reading View's automatic approach. Only 2 people said they would stay with the original page. These results leads to the thinking that most people are not satisfied with the printing of web pages as they are presented on the screen and wish some degree of adaptation to paper.

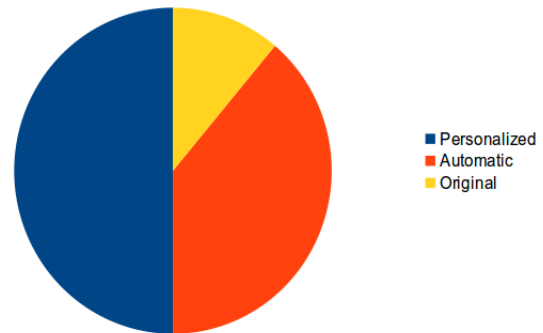


Figure 10: Graph with the interviewee's preference

At the end of the interview we asked the volunteers whether they would use a personalized content extraction application if it was available to them. From the 18 people, 13 said they would use such an application. Some of them said they would use it only if it was made available for a mobile version.

Most people preferred a clean and customizable version. We believe that there are two reasons for this reaction: first, some people really prefer cleaner pages, whereas a second reason could be that people generally wish to be able to customize stuff, even when they do not do that in practice.

6. CONCLUSION

Web pages and specially news pages are in many cases composed by a "main" content usually made of text and images as well as items such as advertisements, menus and others. These may be very distracting to readers and when the page is printed such items usually are sent to the output. This is in most cases a waste of resources since they do not add any value to the content. One solution for this problem is to extract the main content from the page and show it in a cleaner presentation, either on the screen or on paper.

In this work we offer a solution that consists on creating a filter to be applied based on items from the page: Only that content goes through the filter, anything else is blocked and not presented. Users are able to create this filter in a semi-automatic way: They sample elements to be considered relevant and any element similar to the samples is identified and collected by our algorithm. We represent the web pages as the DOM standard, a tree like structure for HTML documents in which each element is a node. Element similarity is measured by comparing their node paths with a variation of the Levenshtein algorithm. Our approach produces a cleaner output containing only elements the users think are interesting.

This approach has the obvious advantage that users provide their own concept of relevance, thus avoiding a possibly complex process of trying to identify relevant items and also allowing for the production of personalized filters better suited to each user.

Also, a filter created for a specific page can be reused for a different page as long as they have a similar structure and the result can be obtained automatically, meaning that

the element sampling is a one-time job that may be used extensively for similar pages.

To validate our approach, we implemented it as a Google Chrome extension and validated its output with users by comparing them with the output of a commercial solution. The users preferred our outputs mainly because it was known to be customizable.

Finally, for future works we intend to detach the filtering implementation from the browser so that it can run as a standalone process, implement the approach for mobile devices and to conduct an user evaluation that examines both the users reaction to the results and their experience using our approach.

7. ACKNOWLEDGMENTS

This paper was achieved in cooperation with Hewlett-Packard Brasil Ltda. using incentives of Brazilian Informatics Law (Law n. 8.2.48 of 1991).

References

[1] Clean Print. http://www.formatdynamics.com/cleanprint-4-0/, 2014. [Online; accessed 24-March-2014].

[2] Evernote Clearly. http://evernote.com/clearly/, 2014. [Online; accessed 24-March-2014].

[3] Internet Explorer Reading View. http://msdn.microsoft.com/en-us/library/ie/hh771832(v=vs.85).aspx#reading-view, 2014. [Online; accessed 24-March-2014].

[4] Reader. http://support.apple.com/kb/ht4550, 2014. [Online; accessed 24-March-2014].

[5] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. Vips: A vision-based page segmentation algorithm. Technical report, Microsoft technical report, MSR-TR-2003-79, 2003.

[6] Suhit Gupta, Gail Kaiser, David Neistadt, and Peter Grimm. Dom-based content extraction of html documents. In *Proceedings of the 12th international conference on World Wide Web*, pages 207–214. ACM, 2003.

[7] HP Clipper. http://www.hpclipper.com/, 2014. [Online; accessed 24-March-2014].

[8] João Batista S. de Oliveira. Two algorithms for automatic document page layout. In *Proceedings of the Eighth ACM Symposium on Document Engineering*, DocEng '08, pages 141–149, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-081-4. doi: 10.1145/1410140.1410170. URL http://doi.acm.org/10.1145/1410140.1410170.

[9] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[10] Suk Hwan Lim, Liwei Zheng, Jianming Jin, Huiman Hou, Jian Fan, and Jerry Liu. Automatic selection of print-worthy content for enhanced web page printing experience. In *Proceedings of the 10th ACM symposium on Document engineering*, pages 165–168. ACM, 2010.

[11] Ping Luo, Jian Fan, Sam Liu, Fen Lin, Yuhong Xiong, and Jerry Liu. Web article extraction for web printing: a dom+ visual based approach. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 66–69. ACM, 2009.

[12] J. Marini. *Document Object Model : Processing Structured Documents: Processing Structured Documents*. McGraw-Hill Professional Publishing, 2002. ISBN 9780072228311. URL http://books.google.com.br/books?id=vFXu8D9ml8AC.

[13] Davi de Castro Reis, Paulo Braz Golgher, ASd Silva, and AF Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web*, pages 502–511. ACM, 2004.

[14] Junfeng Wang, Chun Chen, Can Wang, Jian Pei, Jiajun Bu, Ziyu Guan, and Wei Vivian Zhang. Can we learn a template-independent wrapper for news article extraction from a single training site? In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1345–1354. ACM, 2009.