DOM-based Content Extraction of HTML Documents

Suhit Gupta
Columbia University
Dept. of Comp. Sci.
New York, NY 10027, US
001-212-939-7184
suhit@cs.columbia.edu

Gail Kaiser
Columbia University
Dept. of Comp. Sci.
New York, NY 10027, US
001-212-939-7000
kaiser@cs.columbia.edu

David Neistadt
Columbia University
Dept. of Comp. Sci.
New York, NY 10027, US
001-212-939-7184
dln35@cs.columbia.edu

Peter Grimm
Columbia University
Dept. of Comp. Sci.
New York, NY 10027, US
001-212-939-7184
pmg23@cs.columbia.edu

Abstract

Web pages often contain clutter (such as pop-up ads, unnecessary images and extraneous links) around the body of an article that distracts a user from actual content. Extraction of "useful and relevant" content from web pages has many applications, including cell phone and PDA browsing, speech rendering for the visually impaired, and text summarization. Most approaches to removing clutter or making content more readable involve changing font size or removing HTML and data components such as images, which takes away from a webpage's inherent look and feel. Unlike "Content Reformatting", which aims to reproduce the entire webpage in a more convenient form, our solution directly addresses "Content Extraction". We have developed a framework that employs easily extensible set of techniques that incorporate advantages of previous work on content extraction. Our key insight is to work with the DOM trees, rather than with raw HTML markup. We have implemented our approach in a publicly available Web proxy to extract content from HTML web pages.

1. Introduction

Web pages are often cluttered with distracting features around the body of an article that distract the user from the actual content they're interested in. These "features" may include pop-up ads, flashy banner advertisements, unnecessary images, or links scattered around the screen. Automatic extraction of useful and relevant content from web pages has many applications, ranging from enabling end users to accessing the web more easily over constrained devices like PDAs and cellular phones to providing better access to the web for the visually impaired.

Most traditional approaches to removing clutter or making content more readable involve increasing font size, removing images, disabling JavaScript, etc., all of which eliminate the webpage's inherent look-and-feel. Examples include WPAR [18], Webwiper [19] and JunkBusters [20]. All of these products involve hardcoded techniques for certain common web page designs as well as fixed "blacklists" of advertisers. This can produce inaccurate results if the software encounters a layout that it hasn't been programmed to handle. Another approach has been content reformatting which reorganizes the data so that it fits on a PDA; however, this does not eliminate clutter but merely reorganizes it. Opera [21], for example, utilizes their proprietary Small Screen Rendering technology that reformats web pages to fit inside the screen width. We propose a "Content Extraction" technique that can remove clutter without destroying webpage layout, making more of a page's content viewable at once.

Content extraction is particularly useful for the visually impaired and blind. A common practice for improving web page accessibility for the visually impaired is to increase font size and decrease screen resolution; however, this also increases the size of the clutter, reducing effectiveness. Screen readers for the blind, like Hal Screen Reader by Dolphin Computer Access

or Microsoft's Narrator, don't usually automatically remove such clutter either and often read out full raw HTML. Therefore, both groups benefit from extraction, as less material must be read to obtain the desired results.

Natural Language Processing (NLP) and information retrieval (IR) algorithms can also benefit from content extraction, as they rely on the relevance of content and the reduction of "standard word error rate" to produce accurate results [13]. Content extraction allows the algorithms to process only the extracted content as input as opposed to cluttered data coming directly from the web [14]. Currently, most NLP-based information retrieval applications require writing specialized extractors for each web domain [14][15]. While generalized content extraction is less accurate than hand-tailored extractors, they are often sufficient [23] and reduce labor involved in adopting information retrieval systems.

While many algorithms for content extraction already exist, few working implementations can be applied in a general manner. Our solution employs a series of techniques that address the aforementioned problems. In order to analyze a web page for content extraction, we pass web pages through an open source HTML parser, openXML (http://www.openxml.org), which corrects the markup and creates a Document Object Model tree. The Document Object Model (www.w3.org/DOM) is a standard for creating and manipulating in-memory representations of HTML (and XML) content. By parsing a webpage's HTML into a DOM tree, we can not only extract information from large logical units similar to Buyukkokten's "Semantic Textual Units" (STUs, see [3][4]), but can also manipulate smaller units such as specific links within the structure of the DOM tree. In addition, DOM trees are highly editable and can be easily used to reconstruct a complete webpage. Finally, increasing support for the Document Object Model makes our solution widely portable. We use the Xerces HTML DOM for this project.

2. Related Work

There is a large body of related work in content identification and information retrieval that attempts to solve similar problems using various other techniques. Finn et al. [1] discuss methods for content extraction from "single-article" sources, where content is presumed to be in a single body. The algorithm tokenizes a page into either words or tags; the page is then sectioned into 3 contiguous regions, placing boundaries to partition the document such that most tags are placed into outside regions and word tokens into the center region. This approach works well for single-body documents, but destroys the structure of the HTML and doesn't produce good results for multi-body documents, i.e., where content is segmented into multiple smaller pieces, common on Web logs ("blogs") like Slashdot (http://slashdot.org). In order for content of multi-body documents to be successfully extracted, the running time of the algorithm would become polynomial time with a degree equal to the number of separate bodies, i.e., extraction of a document containing 8 different bodies would run in $O(N^8)$, N being the number of tokens in the document.

McKeown et al. [8][9], in the NLP group at Columbia University, detects the largest body of text on a webpage (by counting the number of words) and classifies that as content. This method works well with simple pages. However, this algorithm produces noisy or inaccurate results handling multi-body documents, especially with random advertisement and image placement.

Rahman et al. [2] propose another technique that uses structural analysis, contextual analysis, and summarization. The structure of an HTML document is first analyzed and then properly decomposed into smaller subsections. The content of the individual sections is then extracted and summarized. However, this

proposal has yet to be implemented. Furthermore, while the paper lays out prerequisites for content extraction, it doesn't actually propose methods to do so.

A variety of approaches have been suggested for formatting web pages to fit on the small screens of cellular phones and PDAs (including the Opera browser [16] that uses the handheld CSS media type, and Bitstream ThunderHawk [17]); however, they basically end up only reorganizing the content of the webpage to fit on a constrained device and require a user to scroll and hunt for content.

Buyukkokten et al. [3][10] define "accordion summarization" as a strategy where a page can be shrunk or expanded much like the instrument. They also discuss a method to transform a web page into a hierarchy of individual content units called Semantic Textual Units, or STUs. First, STUs are built by analyzing syntactic features of an HTML document, such as text contained within paragraph (<P>), table cell (<TD>), and frame component (<FRAME>) tags. These features are then arranged into a hierarchy based on the HTML formatting of each STU. STUs that contain HTML header tags (<H1>, <H2>, and <H3>) or bold text (<B>) are given a higher level in the hierarchy than plain text. This hierarchical structure is finally displayed on PDAs and cellular phones. While Buyukkokten's hierarchy is similar to our DOM tree-based model, DOM trees remain highly editable and can easily be reconstructed back into a complete webpage. DOM trees are also a widely-adopted W3C standard, easing support and integration of our technology. The main problem with the STU approach is that once the STU has been identified, Buyukkokten, et al. [3][4] perform summarization on the STUs to produce the content that is then displayed on PDAs and cell phones. However, this requires editing the original content and displaying information that is different from the original work. Our approach retains all original work.

Kaasinen et al. [5], discusses methods to divide a web page into individual units likened to cards in a deck. Like STUs, a web page is divided into a series of hierarchical "cards" that are placed into a "deck". This deck of cards is presented to the user one card at a time for easy browsing. The paper also suggests a simple conversion of HTML content to WML (Wireless Markup Language), resulting in the removal of simple information such as images and bitmaps from the web page so that scrolling is minimized for small displays. While this reduction has advantages, the method proposed in that paper shares problems with STUs. The problem with the deck-of-cards model is that it relies on splitting a page into tiny sections that can then be browsed as windows. But this means that it is up to the user to determine on which cards the actual contents are located.

None of the concepts solve the problem of automatically extracting just the content, although they do provide simpler means in which the content can be found. Thus, these concepts limit analysis of webpages. By parsing a webpage into a DOM tree, more control can be achieved while extracting content.

3. Our Approach

Our solution employs multiple extensible techniques that incorporate the advantages of the previous work on content extraction. In order to analyze a web page for content extraction, the page is first passed through an HTML parser that creates a DOM tree representation of the web page. We use openXML [22] as our HTML parser, which takes care of correcting the HTML, therefore we do not have to deal with error resiliency. Once processed, the resulting DOM document can be seamlessly shown as a webpage to the end-user as if it were HTML. This process accomplishes the steps of structural analysis and structural decomposition done by Rahman's, Buyukkokten's and Kaasinen's techniques (see Section 2). The DOM tree is hierarchically arranged and can be analyzed in sections or as a

whole, providing a wide range of flexibility for our extraction algorithm. Just as the approach mentioned by Kaasinen et al. modifies the HTML to restructure the content of the page, our content extractor navigates the DOM tree recursively, using a series of different filtering techniques to remove and modify specific nodes and leave only the content behind. Each of the filters can be easily turned on and off and customized to a certain degree.

There are two sets of filters, with different levels of granularity. The first set of filters simply ignores tags or specific attributes within tags. With these filters, images, links, scripts, styles, and many other elements can be quickly removed from the web page. This process of filtering is similar to Kaasinen's conversion of HTML to WML. However, the second set of filters is more complex and algorithmic, providing a higher level of extraction than offered by the conversion of HTML to WML. This set, which can be extended, consists of the advertisement remover, the link list remover, the empty table remover, and the removed link retainer.

The advertisement remover uses an efficient technique to remove advertisements. As the DOM tree is parsed, the values of the "src" and "href" attributes throughout the page are surveyed to determine the servers to which the links refer. If an address matches against a list of common advertisement servers, the node of the DOM tree that contained the link is removed. This process is similar to the use of an operating systems-level "hosts" file to prevent a computer from connecting to advertiser hosts. Hanzlik [6] examines this technique and cites a list of hosts, which we use for our advertisement remover. In order to avoid the common pitfall of deploying a fixed blacklist of advertisers, our software periodically updates the list from http://accs-net.com, a site that specializes in creating such blacklists.

The link list remover employs a filtering technique that removes all "link lists", which are table cells for which the ratio of the number of links to the number of non-linked words is greater than a specific threshold (known as the link/text removal ratio). When the DOM parser encounters a table cell, the Link List Remover tallies the number of links and the number of non-linked words. The number of non-linked words is determined by taking the number of letters not contained in a link and dividing it by the average number of characters per word, which we preset as 5 (although it may be overridden by the user). If the ratio is greater than the user-determined link/text removal ratio, the content of the table cell (and, optionally, the cell itself) is removed. This algorithm succeeds in removing most long link lists that tend to reside along the sides of web pages while leaving the text-intensive portions of the page intact. In Section 4.2, we discuss one of the drawbacks of this technique where link rich pages tend to give undesirable results.

The empty table remover removes tables that are empty of any "substantive" information. The user determines, through settings, which HTML tags should be considered to be substance and how many characters within a table are needed to be viewed as substantive. The table remover checks a table for substance after it has been parsed through the filter. If a table has no substance, it is removed from the tree. This algorithm effectively removes any tables leftover from previous filters that contain small amounts of unimportant information.

While the above filters remove non-content from the page, the removed link retainer adds link information back at the end of the document to keep the page browsable. The removed link retainer keeps track of all the text links that are removed throughout the filtering process. After the DOM tree is completely parsed, the list of removed links is added to the bottom of the page. In this way, any important navigational links that were previously removed remains accessible.

After the entire DOM tree is parsed and modified appropriately, it can be output in either HTML or as plain text. The plain text output removes all the tags and retains only the text of the site, while eliminating most white space. The result is a text document that contains the main content of the page in a format suitable for summarization, speech rendering or storage. This technique is significantly different from Rahman et al. [2], which states that a decomposed webpage should be *analyzed* to find the content. Our algorithm doesn't find the content but eliminates non-content. In this manner, we can still process and return results for sites that don't have an explicit "main body".

4. Implementation

In order to make our extractor easy to use, we implemented it as a web proxy (program and instructions are accessible at http://www.psl.cs.columbia.edu/proxy). This allows an administrator to set up the extractor and provide content extraction services for a group. The proxy is coupled with a graphical user interface (GUI) to customize its behavior. The separate screens of the GUI are shown in figures 1, 2, and 3. The current implementation of the proxy is in Java for cross-platform support.

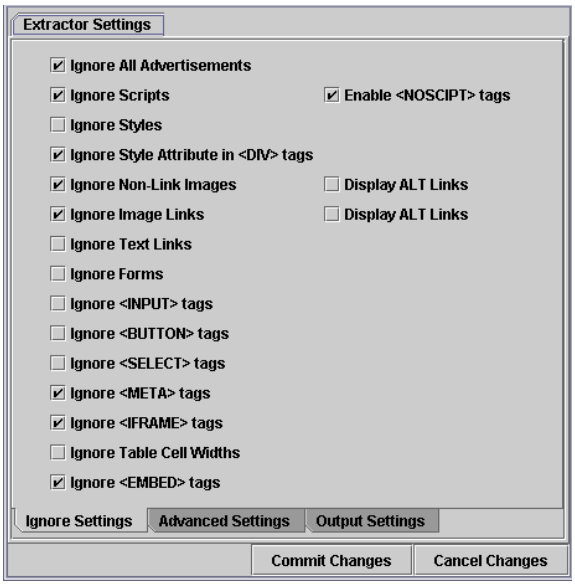

Figure 1

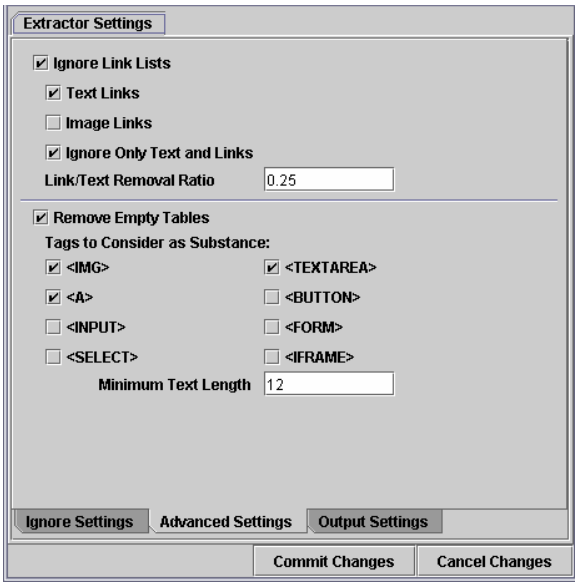

Figure 2

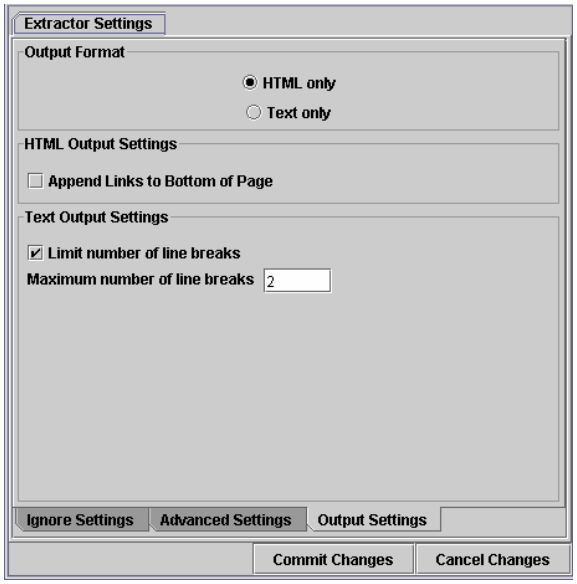

Figure 3

While the Content Extraction algorithm's worst case running time is $O(N^2)$ for complex nested tables; without such nesting, the typical running time is O(N), where N is the number of nodes in the DOM tree after the HTML page is parsed. During tests, the algorithm performs quickly and efficiently following the one-time proxy customization. The proxy can handle most web pages, including those with badly formatted

HTML, because of the corrections automatically applied while the page is parsed into a DOM tree.

Depending on the type and complexity of the web page, the content extraction suite can produce a wide variety of output. The algorithm performs well on pages with large blocks of text such as news articles and mid-size to long informational passages. Most navigational bars and extraneous elements of web pages such as advertisements and side panels are removed or reduced in size. Figures 4 and 5 show an example.

Figure 4 - Before

Figure 5 - After

When printed out in text format, most of the resulting text is directly related to the content of the site, making it possible to use summarization and keyword extraction algorithms efficiently and accurately. Pages with little or no textual content are extracted with varying results. An example of text format extraction performed on the webpage in figure 5 is shown in figure 6.

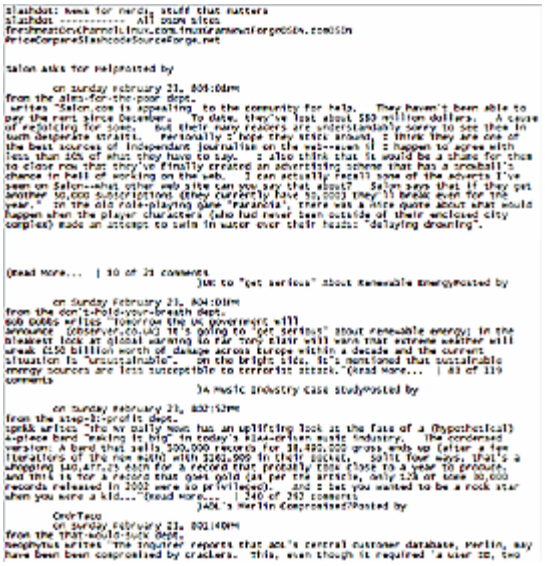

Figure 6

The initial implementation of the proxy was designed for simplicity in order to test and design content extraction algorithms. It spawns a new thread to handle each new connection, limiting its scalability. Most of the performance drop from using the proxy originates from the proxy's need to download the entire page before sending it to the client. Future versions will use staged event architecture and asynchronous callbacks to avoid threading scalability issues.

4.1. Further examples

Figures 7 and 8 show a example of a typical page from www.spacer.com and a filtered version of that page, respectively. This is another good example of a site that is presented in a content-rich format. On the other hand, Figures 9 and 10 show the front page of www.planetunreal.com, a site dedicated to the

Unreal Tournament 2003 first-person shooter game (www.epicgames.com), before and after content extraction. Despite producing results that are rich in text, screenshots of the game are also removed, which the user might deem relevant content.

Figure 7 - Before

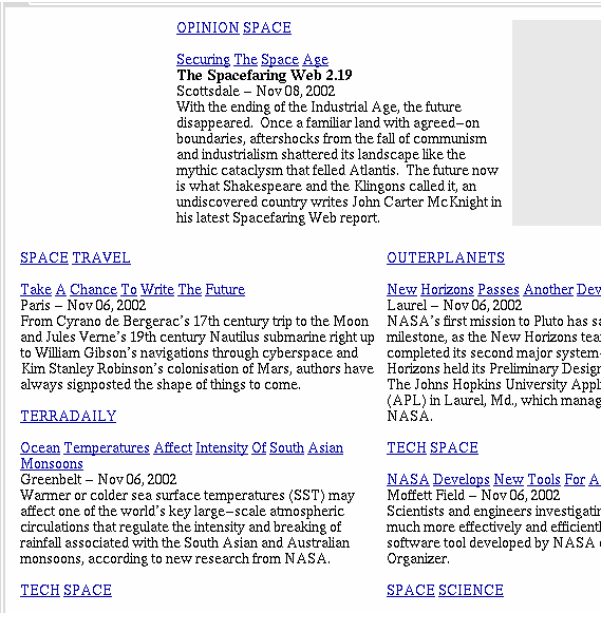

Figure 8 - After

Figure 9 - Before

Figure 10 - After

Figures 11 and 12 show www.msn.com in its pre- and post-filtered state. Since the site is a portal which contains links and little else, the proxy does not find any coherent content to keep. We are investigating heuristics that would leave such pages untouched.

Figure 11 - Before

Figure 12 - After

One may feel from these examples that input fields are affected irregularly by our proxy; however the run-time decision of leaving them in or removing them from the page is dependent on the tables or frames they are contained in. Additionally, we should mention that there isn't any sort of preservation of objects that may be lost after the HTML is passed through our parser. The user would have to change the settings of the proxy and reload the page to see changes.

4.2. Implementation details

In order to analyze a web page for content extraction, the page is passed through an HTML parser that creates a Document Object Model tree. The algorithm begins by starting at the root node of the DOM tree (the <HTML> tag), and proceeds by parsing through its children using a recursive depth first search function called *filterNode()*. The function initializes a Boolean variable (*mCheckChildren*) to true to allow *filterNode()* to check the children. The currently selected node is then passed through a filter method called *passThroughFilters()* that analyzes and modifies the node based on a series of user-selected preferences. At any time within *passThroughFilters()*, the *mCheckChildren* variable can be set to false, which allows the individual filter to prevent specific subtrees from being filtered. After the node is filtered accordingly, *filterNode()* is recursively called using the children if the *mCheckChildren* variable is still true.

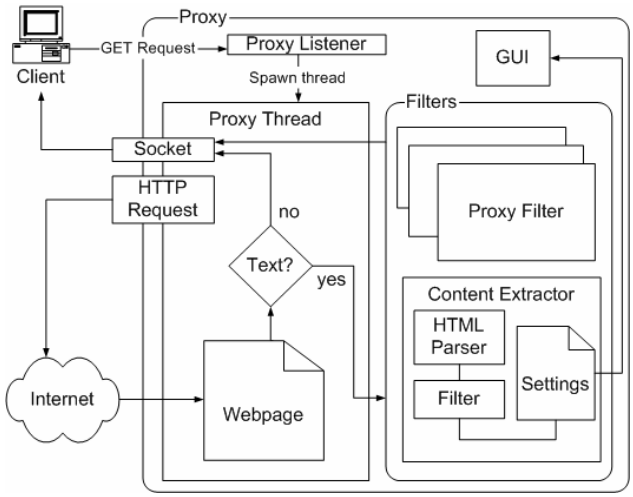

Figure 13. Architectural diagram of the system

The filtering method, *passThroughFilters()*, performs the majority of the content extraction. It begins by examining the node it is passed to see if it is a "text node" (data) or an "element node" (HTML tag). Element nodes are examined and modified in a series of passes. First, any filters that edit an

element node but do not delete it are applied. For example, the user can enable a preference that will remove all table cell widths, and it would be applied in the first phase because it modifies the attributes of table cell nodes without deleting them.

The second phase in examining element nodes is to apply all filters that delete nodes from the DOM tree. Most of these filters prevent the *filterNode()* method from recursively checking the children by setting *mCheckChildren* to false. A few of the filters in this subset set *mCheckChildren* to true so as to continue with a modified version of the original *filterNode()* method. For example, the empty table remover filter sets *mCheckChildren* to false so that it can itself recursively search through the <TABLE> tag using a bottom-up depth first search while *filterNode()* uses a top-down depth first search. Finally, if the node is a text node, any text filters are applied (there are currently none, but there may be in the future).

4.3. Implementation as an integrable framework

Since a content extraction algorithm can be applied to many different fields, we implemented it so that it can be easily used in a variety of cases. Through an extensive set of preferences, the extraction algorithm can be highly customized for different uses. These settings are easily editable through the GUI, method calls, or direct manipulation of the settings file on disk. The GUI itself can also easily be easily integrated (as a Swing JPanel) into any project. The content extraction algorithm is also implemented as an interface for easy incorporation into other Java programs. The content extractor's broad set of features and customizability allow others to easily add their own version of the algorithm to any product.

5. Future Work

The current implementation in Java uses a 3rd-party HTML parser to create DOM trees from web pages. Unfortunately, most of the publicly-available Java HTML parsers are either missing support for important features, such as XHTML or dynamically-generated pages (ASP, JSP). To resolve this, we intend to support commercial parsers, such as Microsoft's HTML parser (which is used in Internet Explorer), in the next revision. These are much more robust and support a wider variety of content. Integration will be accomplished by porting the existing proxy to C#/.NET, which will allow for easy integration with COM components (of which the MS HTML parser is one).

We are also working on improving the proxy's performance; in particular, we aim to improve both latency and scalability of the current version. The latency of the system will be reduced by adopting a commercial parser and by improving our algorithms to process DOM incrementally as the page is being loaded. Scalability will be addressed by re-architecting the proxy's concurrency model to avoid the current thread-per-client model, adopting a stage-driven architecture instead.

Finally, we are investigating supporting more sophisticated statistical, information retrieval and natural language processing approaches as additional heuristics to improve the utility and accuracy of our current system.

6. Conclusion

Many web pages contain excessive clutter around the body of an article. Although much research has been done on content extraction, it is still a relatively new field. Our approach, working with the Document Object Model tree as opposed to raw HTML markup, enables us to perform Content Extraction, identifying and preserving the original data instead of summarizing it. The techniques that we have employed, though simple, are quite effective. As a result, we were able to implement our approach in a publicly-available proxy that anyone can use to extract content from web pages for their own purposes.

7. Acknowledgements

The Programming Systems Laboratory is funded in part by Defense Advanced Research Project Agency under DARPA Order K503 monitored by Air Force Research Laboratory F30602-00-2-0611, by National Science Foundation grants CCR-02-03876, EIA-00-71954, and CCR-99-70790, and by Microsoft Research and IBM.

In addition, we would like to extend a special thanks to Janak Parekh, Philip N. Gross and Jean-Denis Greze.

8. References

[1] Aidan Finn, Nicholas Kushmerick and Barry Smyth. "Fact or fiction: Content classification for digital libraries". In Joint DELOS-NSF Workshop on Personalisation and Recommender Systems in Digital Libraries (Dublin), 2001.

[2] A. F. R. Rahman, H. Alam and R. Hartono. "Content Extraction from HTML Documents". In 1st Int. Workshop on Web Document Analysis (WDA2001), 2001.

[3] O. Buyukkokten, H. Garcia-Molina and A. Paepcke. "Accordion Summarization for End-Game Browsing on PDAs and Cellular Phones". In Proc. of Conf. on Human Factors in Computing Systems (CHI'01), 2001.

[4] O. Buyukkokten, H, Garcia-Molina and A. Paepcke. "Seeing the Whole in Parts: Text Summarization for Web Browsing on Handheld Devices". In Proc. of 10th Int. World-Wide Web Conf., 2001.

[5] E. Kaasinen, M. Aaltonen, J. Kolari, S. Melakoski and T. Laakko. "Two Approaches to Bringing Internet Services to WAP devices". In Proc. of 9th Int. World-Wide Web Conf., 2000.

[6] Stuart Hanzlik "Gorilla Design Studios Presents: The Hosts File". Gorilla Design Studios. August 31, 2002. http://accs-net.com/hosts/.

[7] Marc H. Brown and Robert A. Shillner. "A New Paradigm for Browsing the Web". In Human Factors in Computing Systems (CHI'95 Conference Companion), 1995.

[8] K.R. McKeown, R. Barzilay, D. Evans, V. Hatzivassiloglou, M.Y. Kan, B. Schiffman and S. Teufel. "Columbia Multi-document Summarization: Approach and Evaluation", In Document Understanding Conf., 2001.

[9] N. Wacholder, D. Evans and J. Klavans. "Automatic Identification and Organization of Index Terms for Interactive Browsing". In Joint Conf. on Digital Libraries '01, 2001.

[10] O. Buyukkokten, H. Garcia-Molina and A. Paepcke. "Text Summarization for Web Browsing on Handheld Devices", In Proc. of 10th Int. World-Wide Web Conf., 2001.

[11] Manuela Kunze and Dietmar Rosner. "An XML-based Approach for the Presentation and Exploitation of Extracted Information". In 19th International Conference on Computational Linguistics, (Coling) 2002.

[12] A. F. R. Rahman, H. Alam and R. Hartono. "Understanding the Flow of Content in Summarizing HTML Documents". In Int. Workshop on Document Layout Interpretation and its Applications, DLIA01, Sep., 2001.

[13] Wolfgang Reichl, Bob Carpenter, Jennifer Chu-Carroll and Wu Chou. "Language Modeling for Content Extraction in Human-Computer Dialogues". In Int. Conf. on Spoken Language Processing, 1998.

[14] Ion Muslea, Steve Minton and Craig Knoblock. "A Hierarchal Approach to Wrapper Induction". In Proc. of 3rd Int. Conf. on Autonomous Agents (Agents'99), 1999.

[15] Min-Yen Kan, Judith L. Klavans and Kathleen R. McKeown. "Linear Segmentation and Segment Relevance". In Proc. of 6th Int. Workshop of Very Large Corpora (WVLC-6), 1998.

[16] http://www.opera.com

[17] http://www.bitstream.com/wireless

[18] http://sourceforge.net/projects/wpar

[19] http://www.webwiper.com

[20] http://www.junkbusters.com

[21] http://www.opera.com

[22] http://www.openxml.org

[23] Private communication, Min-Yen Kan, Columbia NLP group, 2002.