Extracting Context To Improve Accuracy For HTML Content Extraction

Suhit Gupta
Columbia University
500 W. 120th Street
New York, NY 10027
001-212-939-7184
suhit@cs.columbia.edu

Gail Kaiser
Columbia University
500 W. 120th Street
New York, NY 10027
001-212-939-7081
kaiser@cs.columbia.edu

Salvatore Stolfo
Columbia University
500 W. 120th Street
New York, NY 10027
001-212-939-7080
sal@cs.columbia.edu

ABSTRACT

Web pages contain clutter (such as ads, unnecessary images and extraneous links) around the body of an article, which 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, reducing noise for information retrieval systems and to generally improve the web browsing experience. In our previous work [16], we developed a framework that employed an easily extensible set of techniques that incorporated results from our earlier work on content extraction [16]. Our insight was to work with DOM trees, rather than raw HTML markup. We present here filters that reduce human involvement in applying heuristic settings for websites and instead automate the job by detecting and utilizing the physical layout and content genre of a given website. We also present work we have done towards improving the usability and performance of our content extraction proxy as well as the quality and accuracy of the heuristics that act as filters for inferring the context of a webpage.

Categories and Subject Descriptors

I.7.4 [**Document and Text Processing**]: Electronic Publishing; H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *Web-based Services*

General Terms

Human Factors, Algorithms, Standardization.

Keywords

DOM trees, content extraction, reformatting, HTML, context, accessibility, speech rendering.

1. INTRODUCTION

Users are spending more and more time on the Internet in today's world of online shopping and banking; meanwhile, webpages are getting more complex in design and content. Web pages are cluttered with guides and menus attempting to improve the user's efficiency, but they often end up distracting from the actual content of interest. These "features" may include script- and flash-driven animation, menus, pop-up ads, obtrusive banner advertisements, unnecessary images, or links scattered around the screen. The automatic extraction of useful and relevant content from web pages has many applications, including enabling end users to access the web more easily over constrained devices like PDAs and cellular phones, providing better access to the web for the visually impaired, providing less noisy data for information retrieval and summarization algorithms, and generally improving the web surfing experience.

Content extraction is particularly useful for the visually impaired and the blind. A common practice for improving webpage accessibility for the visually impaired is to increase font size and decrease screen resolution; however, this also increases the size of clutter, reducing efficiency. Screen readers for the blind, like Hal Screen Reader, Microsoft's Narrator or IBM Homepage Reader generally don't remove such clutter either and often read out raw HTML. 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 such algorithms to process only the extracted content, instead of either using cluttered data from the web, or writing specialized extractors for each web domain [14][15].

Other traditional approaches to removing clutter or making content more readable include removing images, disabling JavaScript, etc., all of which eliminate the webpage's original 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. There have also been multiple approaches suggested for formatting web pages to fit on the small screens of cellular phones and PDAs (including the Opera browser [16] and its use of the handheld CSS media type, and Bitstream ThunderHawk [17]); however, such techniques reorganize and reformat the content of the webpage to fit on a constrained device and require a user to scroll and hunt for content.

Our solution employs a series of techniques that addresses the aforementioned problems and creates a simple solution usable and customizable by an end-user. In order to analyze a web page for content extraction, we pass web pages through an HTML parser, 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.

We found DOM trees to be highly editable and easily renderable as a complete webpage. Increasing support for the Document Object Model also makes our solution widely portable. This technique was used to create our initial proof of concept version, Crunch, which is an open ended framework for integrating content extraction heuristics developed by ourselves and others. Crunch is created as a web proxy usable with arbitrary browsers and assistive technologies. It is customizable by an administrator or user to toggle individual heuristics in order to produce the best results. Our initial work [16] showed that we were able to achieve good results but that we also had several limitations. We have since made our proxy more robust in terms of performance, created a multi-pass filtering mechanism, improved the user interface and further added support for various scripted pages as well as for cascading style-sheets (CSS).

One problem with content extraction in general is that it is impossible to determine the intention of the author and the desires of the reader. Therefore our goal is to approach the problem heuristically and work towards making our heuristics as accurate as possible. Crunch extracts the "content", with filters customizable by an administrator and/or by a savvy user. No "one size fits all" algorithm could achieve this goal. In particular, we did not attempt to model either author or user tasks, nor their corresponding context or intentions, but any non-intrusive approach to doing so would also likely be heuristic and thus also imprecise. Therefore, one of the limitations of our framework was that Crunch could potentially remove items from the web page that the user may be interested in, and may present content that the user is not particularly interested in.

We have addressed this by employing a multi-pass filtering system where the resulting DOM tree produced after each stage of filtering is compared to the original copy. If too much or too little has been removed, given the settings, we assume that the settings were set incorrectly and fix them during the next pass over the DOM. We also try to determine the classification of a given website, both in its physical layout as well as the *context* of its content. We have found that, given a manually-created frequently updated database of preset heuristic settings for different genres of websites, we can use this contextual information to dynamically utilize matching settings from our database and produce better results of extracted webpage content.

In this paper, we explain the improvements we have made to Crunch, and explain how we can dynamically categorize sites by determining layout structure and the content context of websites. As we will describe, our solution enables dynamic filtering for a wider range of websites. The following sections briefly describe the background, our approach to the problem, related work and implementation details, and we end with future work directions and a conclusion.

2. BACKGROUND

Crunch 1.0 [16] demonstrated the design of the system as a viable framework, but certain problems needed to be addressed for the system to be widely usable. After releasing Crunch 1.0 in September 2002, we received several suggestions from early users for additions and improvements. These ranged from changes in the aggressiveness of content removal by the filters, to the way our proxy reassembled a webpage from its constituent parts. We conducted an informal user study of blindfolded students in May 2003, followed by a formal user study with blind and visually impaired users, conducted in December 2003 [17]. The NLP group at Columbia University evaluated Crunch as an input mechanism to their Newsblaster [8][9] project, which is a system that automatically tracks, clusters and summarizes each day's news programmatically. They found results to be encouraging, especially since it required little prior training, and by utilizing Crunch they ran their natural language processing algorithms on content extracted by Crunch rather than noisy data streams coming straight from the web.

Crunch 2.0 [17] was similar to its predecessor in that it utilized the DOM model and was also a filter plug-in based framework. We improved its performance, user interface, and heuristic filters. With the original version, we had problems filtering pages with frames or too many links, where the filters would either remove too much, resulting in a blank page, or too little, resulting in a mostly untouched page. Therefore, we implemented a multi-pass filtering system for the heuristics that re-evaluates the modified DOM tree after each pass. After each phase, the produced DOM tree is compared to the previous version, and that phase's results are discarded if the change between the two is either insignificant or too drastic (given the settings). This prevents link-heavy pages like www.msn.com from returning blank pages as output. We improved the performance of the content extractor filter while maintaining its functionality. Additional filters were also added that allow the user to control the font size and word wrapping of the output.

Crunch 3.0, presented in this paper, automates the application of filter settings for a varied range of websites by detecting content genres and the physical layout. We explain these improvements in greater detail in the following sections.

3. CONTEXT EXTRACTION

We find that with the DOM-based content extraction of the earlier version of Crunch (see section 5 for implementation details), we are able to achieve high levels of content extraction accuracy for a wide domain of websites. Example output screenshots shown in our original paper [16] and the subsequent journal paper [17] demonstrated that while the results that our proxy produced (with optimal settings) were quite accurate, the settings for the various filters often had to be tweaked by hand by the administrator or the end user for websites that differed in context and layout. For example, if a user were to browse a typical news-based site, e.g., CNN, then their settings would remove heavy links, images with links (menus), advertisements, and forms, since they would typically be looking for only the articles contained in each page. The results produced, as shown in Figure 1, would be expected. However, if the user chose to switch workflow contexts and browse Amazon.com instead – a link-heavy page with lots of advertisements, images and forms – those important links would be lost due to the news-optimized settings, resulting in an undesirable result as shown in Figure 2. This would easily be fixed by adjusting the settings in the Content Extraction filter of our proxy; however it would require the user to switch application focus and do so manually. The user would have throttled the link/text ratio higher, perhaps even toggled off the advertise remover and the scripts remover (Figure 2) for a better shopping experience.

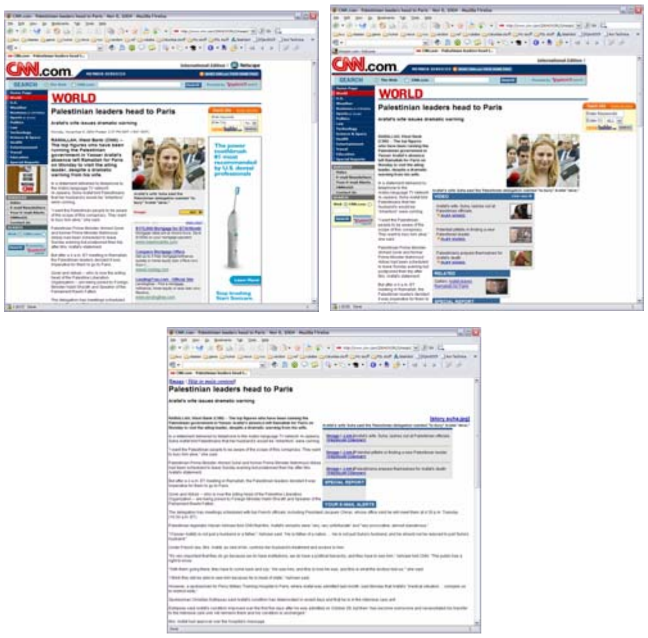


Figure 1 - CNN (original, through Crunch 2.0 with shopping settings and through Crunch 3.0 with auto-settings)

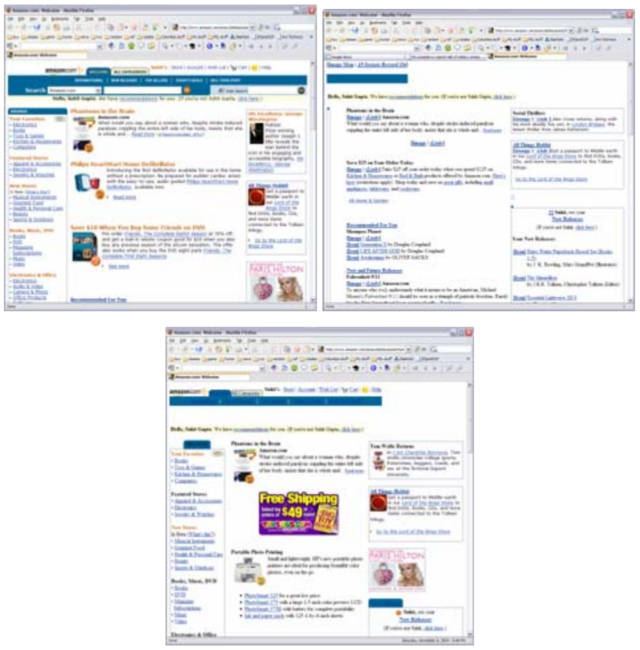


Figure 2 - Amazon (original, through Crunch 2.0 with news settings and through Crunch 3.0 with auto-settings)

As explained before, this workflow results in poor user productivity. Since creating a set of filter settings for producing good results for known webpages is straightforward, we focused on classifying websites. Ideally, a website's classification would be matched against known classes and an appropriate configuration would be selected. For example, if the user browsing CNN were to then browse Amazon.com, for which a classification didn't already exist, Crunch would detect that Amazon.com is a shopping site, similar to others already categorized, and those settings would be adopted automatically. This eliminates the need for human intervention every time there is a context switch on the part of the user.

We have chosen two types of classification to categorize websites – *physical layout* and *content genre*. Our approach to classifying websites based on genre involves a one-time initial processing stage where we pre-classify 200 of the top sites visited by our group on a daily basis. We combine the textual content of the site itself with the contents of search results returned by searching for the site's domain name on three of the top Internet search engines (Google, Yahoo and Dogpile). Adding text from search engine results enables us to leverage the small blurb describing the site in the engine's results, which assists in classifying the site. Combining results per domain also improves the frequency of the occurrence of words that describe the function and content of the site. We then remove all stop words and count the frequency of the words in the remaining text. Since the resulting 200 sets of frequency graphs contain both words that repeat across graphs and words that occur too infrequently to affect content information, we pick all unique words that occur in at least one of the graphs more than five times, creating a master set of *Key Words*. A re-graph against this new set of Key Words produces an accurate content genre *identifier* for each of these websites (examples of results for Amazon and eBay are shown in Figures 3 and 4 respectively).

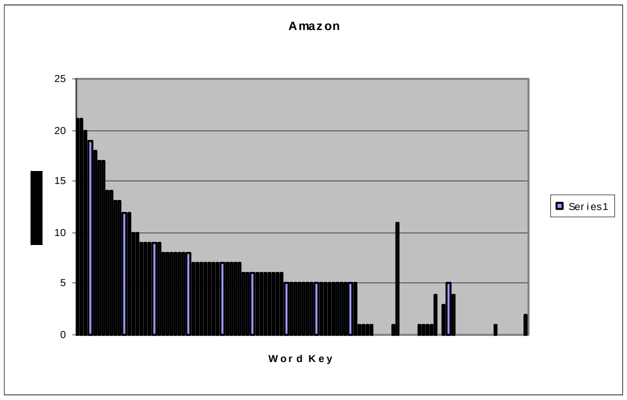


Figure 3 - Frequency chart for Amazon

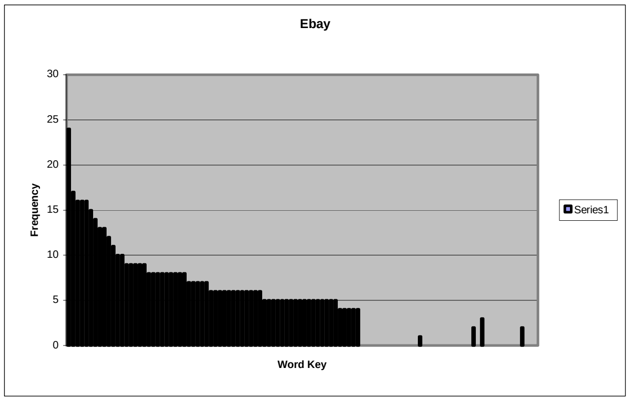


Figure 4 - Frequency Chart for eBay

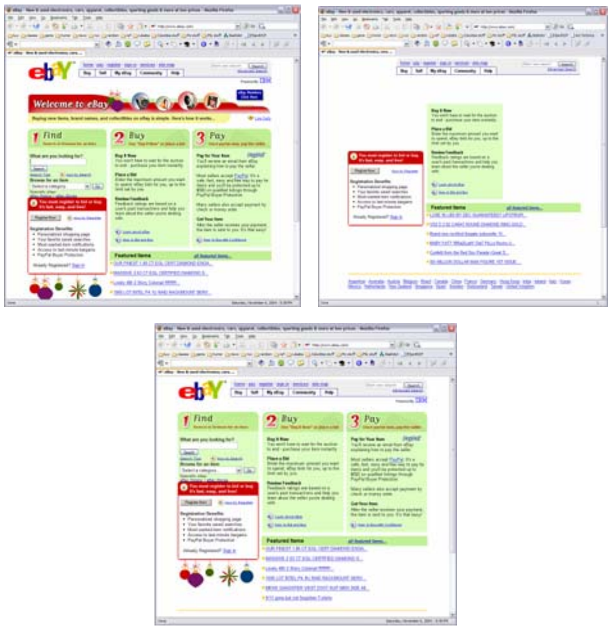


Figure 5 - eBay (original, through Crunch 2.0 with news settings and through Crunch 3.0 with auto-settings)

We can now use our classifications in conjunction with a database containing filter settings for each of the 200 sites. When we encounter a website that we haven't seen before, we again extract the text from the site and corresponding search engine blurb, and perform a frequency match against the frequently-occurring Key Words. We then use the *Manhattan histogram distance measure algorithm* to measure the distance between the website in question and our original classifications. The formula is defined as

$$D_1(h_1,h_2)=\sum_{i=0}^{n-1}|h_1[i]-h_2[i]|$$

The histogram (h_1,h_2) is represented as a vector, where n is the number of bins in the histogram (i.e., the number of words in our Key Word Set). h_1 and h_2 must first be normalized in order to satisfy the above distance function requirements. In Crunch, the sum of the histogram's bins is normalized to 1 before computing the distance. We use the settings associated with the website whose distance is closest to the one being accessed.

In our previous example with CNN, Amazon and eBay, the user might have already defined preferences for CNN and Amazon. If the user navigates to eBay, the histogram matching algorithm finds it to be most similar to Amazon from our pool of categorized sites and picks the equivalent settings. We find that the resulting page (Crunch 3 result - Figure 5) is quite acceptable.

We also tried other approaches and compared their accuracy and speed to the Manhattan distance. For example, Euclidean distance measuring gave us equivalent results with slightly greater computational overhead.

$$D_2(h_1,h_2)=\sum_{i=0}^{n-1}(h_1[i]-h_2[i])^2$$

We have also tried using a simplified version of the Mahalanobis histogram distance formula

$$d^2(x,\overline{y})=(x-\overline{y})^T C^{-1}(x-\overline{y})$$

where x and $\overline{y}$ are two feature vectors, and each element of the vector is a variable. x is the feature vector of the new observation, y is the averaged feature vector computed from the training examples, and C^{-1} is the inverse covariance matrix, where

$$C_{ij}=Cov(y_i,y_j).y_i,y_j$$

are the ith and jth elements of the training vector. The advantage of Mahalanobis distance is that it takes into account not only the average value but also its variance and the covariance of the variables measured. We expected the Mahalanobis formula to be most accurate; however, this was not the case, probably due to the lack of a large amount of variance in our training data. If we had results from more then three different search engines, we could potentially improve the variance in our data and the Mahalanobis histogram distance might give us more reliable results. The current system uses the simple and efficient Manhattan histogram distance measure.

In Figures 6 and 7, we show the frequency chart for CNN and Spacer.com (an astronomy news related site) respectively. Crunch determines that both these sites are similar enough to warrant the same set of settings. The results viewed in the browser can be seen in the Crunch 3.0 results of Figure 8.

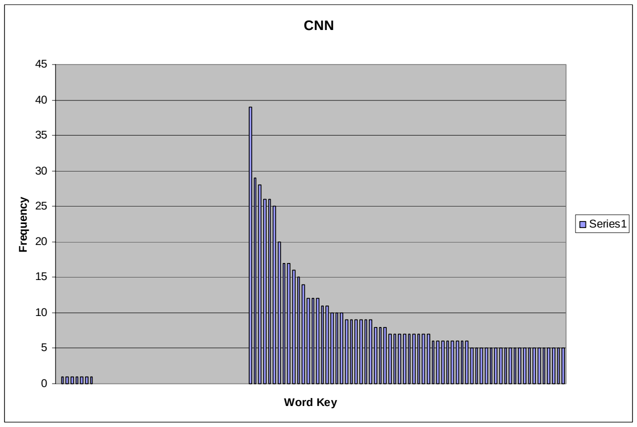


Figure 6 - Frequency chart for CNN

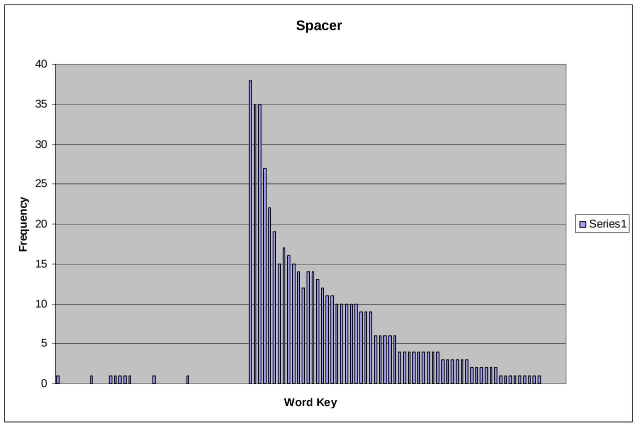


Figure 7 - Frequency chart for Spacer

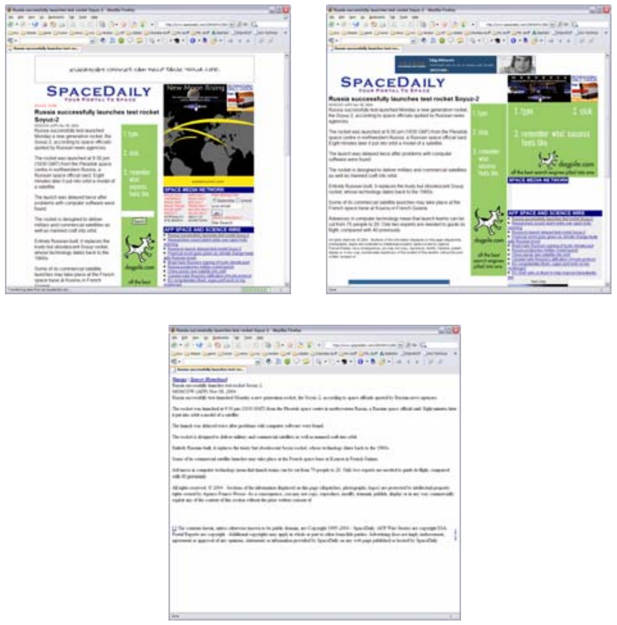


Figure 8 - Spacer.com (original, through Crunch 2.0 with shopping settings and through Crunch 3.0 with auto-settings)

As shown in Figures 4-8, we do not classify websites exclusively into predefined genres. Based on the word frequency histogram we either find an already-defined genre with a similar histogram or create a new genre centered on the new website. Currently websites are assumed to be identical within the domain, which works well for homogeneous sites created with a content management system [39], but less well for heterogeneous sites such as geocities.com assumed to be of a single genre for the entire domain.

In order to classify websites based on their physical characteristics, we manually created a database of physical layouts of the same 200 top visited sites, and classified them into several basic categories. Categorizations include the number of columns, link density, type of site (news, shopping, banking, Weblog, etc.) and the percentage amount of "content" data contained in the various columns. Future work includes automating this process. Our approach will be to analyze the constituent parts of the HTML in the DOM tree using tree pattern inference. [38]

The user has the option of overriding our automatic settings through Crunch's user interface. Ultimately, we hope to use the data produced using physical classification to add to the information gained through content and genre classification to produce even more accurate results.

4. RELATED WORK

4.1 Content Extraction

There is a large body of related work in content identification and information retrieval that attempts to solve similar problems using various other techniques. However, we have found that most of these solutions are too time consuming to be effective for web browsing. Rahman et al. [2] propose techniques that use structural analysis, contextual analysis, and summarization. The structure of an HTML document is first analyzed and then decomposed into smaller subsections. The content of the individual sections is then extracted and summarized. While the paper describes prerequisites for content extraction, it doesn't propose methods to do so. The solution is meant for constrained devices like cell phones, but the user has little control over the output that s/he views. Their technique is not adjustable; therefore the user has low flexibility retrieving removed content.

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 exponential 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][15], 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 when handling multi-body documents, especially with random advertisement and image placement.

Multiple approaches have been suggested for formatting web pages to fit on the small screens of cellular phones and PDAs (including the Opera browser [16] and its use of the handheld CSS media type, and Bitstream ThunderHawk [17]); however, the reformatting approaches generally do not distinguish significant from subsidiary content (that is, clutter), nor remove the latter.

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. This requires very time consuming processing on the original content.

Kaasinen et al. [5] discuss 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.

4.2 Genre Classification

There is some related work that tries to classify webpages by genre which we found to be helpful. Karlgren et al. [33][34] showed that the texts that were judged relevant to a set of queries differ substantially from the texts that were not relevant. Stamatatos et al. [29] show that the frequency of word occurrence is very useful in automatic text genre classification. This approach is similar to ours, and produces results that are domain independent and require minimal computation.

There are approaches that detect genre based on surface cues where comparisons are made between the performances of function words and the Parts Of Speech trigrams. Kessler et al. [30] and Argamon et al. have shown good results with this technique; however, their approach is dependent on substantial bodies of text, the domain of their classification is fairly limited, and cannot be applied dynamically to all web-sites.

Roussinov et al. [32] define genre as a group of documents with similar form, topic or purpose, "a distinctive type of communicative action, characterized by a socially recognized communicative purpose and common aspects of form". They show the advantages of browsing the web by genre but their application is only designed to help categorize documents so users can see similar pages. There is no work in terms of content extraction.

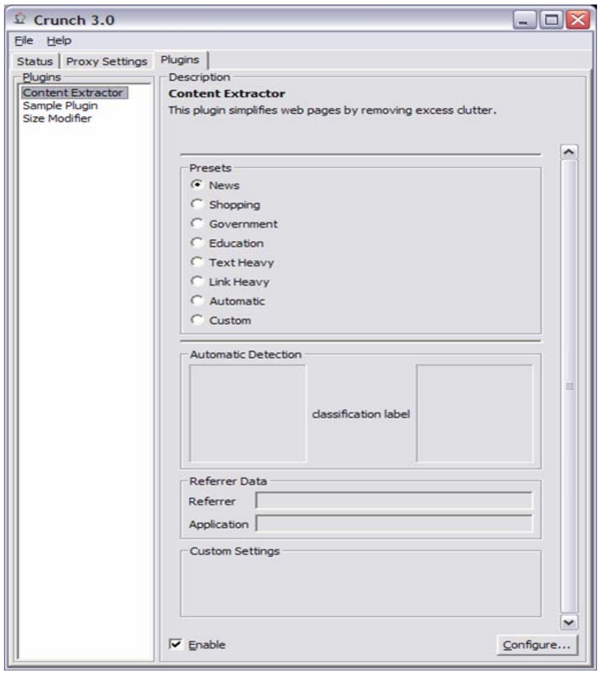
Rauber et al. [36] use the age of a document and frequency of lookups as important distinguishing features and present a method of automatic analysis based on various surface level features of the text.

While all these approaches are valid, we find them all to be lacking for our problem domain. Most are either too computationally expensive or are too domain specific. We also find that most approaches try to extract a word or phrase to describe the context of a given page. We instead compute the genre from a large set of words with individual weights, as it allows for greater accuracy when comparing between an extremely varied ranges of websites.

5. IMPLEMENTATION DETAILS

In the new version of Crunch, we have improved flexibility for most filters by improving the plug-in API as well as by adding new features to each. Users now have the ability of controlling, at a finer granularity, the filtering of complex web pages where certain HTML structures are embedded within others, e.g., controlling not only the content on the entire page but also controlling the parameters that address table cells.

Figures 10 and 11 show some of these changes. Additional example screenshots, including a high contrast scheme for vision-impaired users, are contained in the Appendix at the end of the paper.



Plugins: Content Extractor; Sample Plugin; Size Modifier

Description
Content Extractor
This plugin simplifies web pages by removing excess clutter.

Presets: [✓ News] Shopping Government Education Text Heavy Link Heavy Automatic Custom

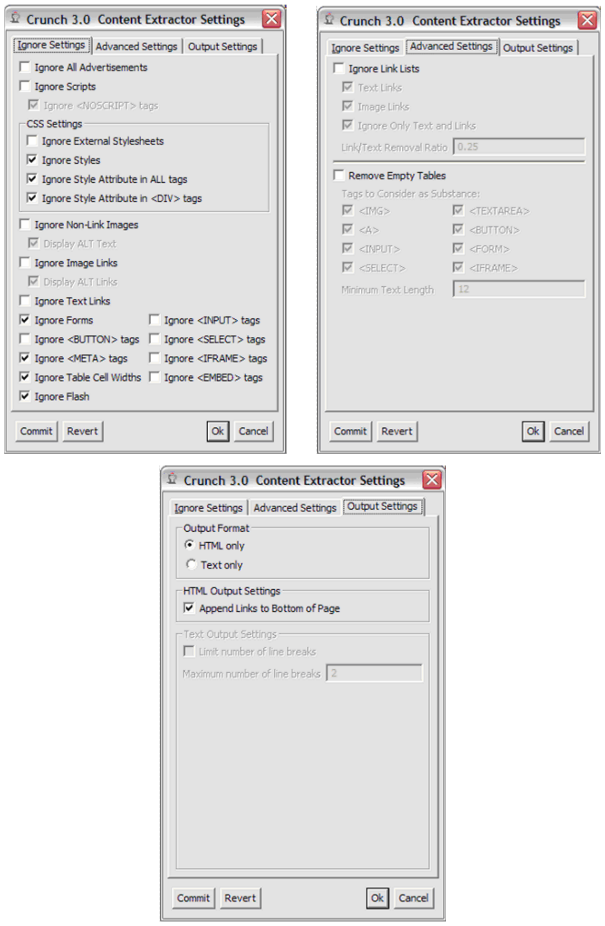
Automatic Detection: classification label

Referrer Data: Referrer; Application

Custom Settings

☑ Enable

Figure 9 - Crunch 3.0 Plug-ins Window


Crunch 3.0 Content Extractor Settings
Ignore Settings | Advanced Settings | Output Settings
Commit Revert Ok Cancel

Ignore Settings:
☐ Ignore All Advertisements
☐ Ignore Scripts
☑ Ignore <NOSCRIPT> tags
CSS Settings
☐ Ignore External Stylesheets
☑ Ignore Styles
☑ Ignore Style Attribute in ALL tags
☑ Ignore Style Attribute in <DIV> tags
☐ Ignore Non-Link Images
☑ Display ALT Text
☐ Ignore Image Links
☑ Display ALT Links
☐ Ignore Text Links
☑ Ignore Forms ☐ Ignore <INPUT> tags
☐ Ignore <BUTTON> tags ☐ Ignore <SELECT> tags
☑ Ignore <META> tags ☐ Ignore <IFRAME> tags
☑ Ignore Table Cell Widths ☐ Ignore <EMBED> tags
☑ Ignore Flash

Advanced Settings:
☐ Ignore Link Lists
☑ Text Links
☑ Image Links
☑ Ignore Only Text and Links
Link/Text Removal Ratio 0.25
☐ Remove Empty Tables
Tags to Consider as Substance:
☑ <IMG> ☑ <TEXTAREA>
☑ <A> ☑ <BUTTON>
☑ <INPUT> ☑ <FORM>
☑ <SELECT> ☑ <IFRAME>
Minimum Text Length 12

Output Settings:
Output Format: [✓ HTML only] Text only
HTML Output Settings
☑ Append Links to Bottom of Page
Text Output Settings
☐ Limit number of line breaks
Maximum number of line breaks 2

Figure 10 - Crunch 3.0 Settings Windows

Like with the previous versions of Crunch, the average runtime complexity of the newer version remains at O(N+P), where N is the number of nodes in the DOM tree after the HTML page is parsed and P is the big-O complexity of the least-efficient plug-in. However, the worst case running time increases to O(N*P) due to our addition of the multi-pass system: in case of a bad result, a filtered webpage may have to revert to a previous state and re-run through the proxy with a different set of options, and this may happen for any number of nodes in the DOM tree. [17]

We have also added support for CSS files and script-generated webpages. Previously, Crunch did not understand CSS, and it stripped away all such non-HTML content from a webpage. This often led to pages that had lost their original look and feel. We now preserve CSS alongside HTML, producing a page that largely retains the appearance of the original site. The user still has the ability to apply their own CSS directives should their web browser support it, since that is a client-side operation, and Crunch simply acts as a proxy. Additionally, we do not strip out Javascript or other embedded script tags out of the HTML content in our first pass. Both these features have allowed us to support a much wider variety of websites.

As pointed out before, we have spent a fair amount of time on improving Crunch's usability. Other implementation changes include:

1) The original version of Crunch used OpenXML [25] as its HTML parser. As we noted in our previous work, OpenXML has efficiency problems that are unlikely to be fixed since OpenXML is apparently no longer an active project. Instead, we switched to NekoHTML [35] – an HTML scanner, tag balancer and parser for the Apache open-source project, Xerces [35]. We chose this new parser as it has many benefits – most notably, increased parsing speed and robust correction of buggy HTML/XML. One of the key longer-term benefits is that we are now using a parser that is under active development. NekoHTML currently has some problems parsing some pages; in particular, the output is not always rendered the same as input, e.g., certain complex nested tables and some CSS-enabled pages. [17] However, most of these errors are minor cosmetic ones that Crunch usually manages to fix in its new multi-pass scheme. Additionally, the developers of NekoHTML are working on its deficiencies. NekoHTML assists in our handling of multiple versions of HTML. Much like our work with the previous parser, Crunch downloads the appropriate HTML stream and sends it to NekoHTML and gets back a DOM tree upon which it applies filters. It then uses an HTML serializer to send data to the client. With this architecture, Crunch can handle any version of HTML NekoHTML supports, including all current versions of HTML and xHTML. [17]

2) Since users often get frustrated with the latency in loading of webpages, we spent a substantial amount of time tuning the performance of the proxy to produce more near-real time results. Some speed improvement was achieved through switching to NekoHTML. The other major contributor to increased speed was the optimization of Crunch's networking code, originally written using Java's blocking I/O API. By collapsing multiple writes and reads, dealing with timeouts more efficiently, and removing unnecessary or redundant calls in the transfer loops, server performance and bandwidth utilization now seems adequate.

In order to try and deal with large workgroup loads, we have migrated Crunch to a staged event architecture using Java's non-blocking I/O API. We now use asynchronous callbacks to avoid threading scalability issues. The concept of a staged event architecture was introduced formally by Welsh [37] for performance gains in highly concurrent server applications, so that they are able to "support massive concurrency demands" [37]. The concept of thread pools helps large-scale systems like Apache webserver deal with the load spikes efficiently. We took the same concept and extended it in our framework so that Crunch can meet the demands of several parallel requests in a groupware setup. We have not yet conducted a performance study with large loads, but we hope to soon.

3) One of the biggest changes made in recent versions of Crunch was to change the basis of the user interface from Java Swing to IBM's SWT (Standard Widget Toolkit [16]). The original UI made our proxy sluggish and user unfriendly. SWT has an extremely clean interface, allows the creation of attractive UIs, and is highly responsive, partially due to its use of JNI and native routine calls that can take advantage of the operating system's built-in optimizations. It also uses native GUI widgets to provide a look and feel consistent with the operating system, while remaining operating system independent. As an added benefit, SWT allows the program to be compiled into a binary executable, resulting in a faster startup time, a smaller distribution, less memory utilization, and an easier installation for novice users. The latest version of Crunch can be downloaded in executable form from our website at http://www.psl.cs.columbia.edu/crunch.

Screenshots of the new proxy GUI are shown above as well as in the Appendix, where we see the basic settings and the available plug-ins as well as the advanced functions.

4) Accessibility was another focus. Switching to SWT helped us maximize accessibility, as described below. One of Crunch's main goals is to assist disabled persons in browsing the web, yet the previous version of Crunch's UI was highly inaccessible. Visually impaired users were then dependent on an administrator to adjust their settings. There are millions of blind or visually impaired people in the US alone and only a small fraction of them are currently able to surf the web [17]. Worse, visually impaired users will often spend several minutes in finding the content they are seeking on a given website. We conducted a small user study with Dr. Michael Chiang, M.D., Instructor in Clinical Ophthalmology and a Research Master's Candidate in the Department of Medical Informatics here at Columbia University. The goal was to find the common causes of visually impaired users' browsing latency to help us address them directly. Our goal was to use Crunch to reduce this time to something more in line with regular users.

There are three basic categories of accessibility support: mobility enablement, visual enhancement, and screen readers [17]. Crunch provides mobility enablement as all settings can be easily accessed using a keyboard in lieu of a mouse. SWT provides keyboard accelerators in the API and supports intelligent tabbing through GUI components. SWT leverages the operating system's accessibility support [17]; therefore Windows' ability to use large fonts and high-contrast themes works with Crunch. SWT also supports Microsoft Active Accessibility Support (MAAS) and its associated widgets, so Crunch automatically supports screen readers that read content from the window with focus. An example screenshot of such changes is in the Appendix.

6. FUTURE WORK

Detecting bad output, and especially quantifying what makes bad output bad, is a hard problem. Training our multi-pass filter to learn to detect bad output is one of our major goals for the future.

We have found that sites that require large amount of form-based input, like most shopping and/or banking sites, often need to be rendered in a specific format in order for them to be usable by the user. Currently, we only have a binary option of removing forms or leaving them in. We plan to work on detecting related form items and showing those that are most relevant to the task at hand.

While NekoHTML works well, we would also like to test a commercial HTML parser like the ones that are bundled with Mozilla or Internet Explorer and compare their efficiency.

We will also continue to work on improving the latency and scalability of Crunch, especially since tabbed browsing is becoming increasingly popular and users often open up several tens of pages at a time.

We would also like to create physical layout heuristics by inferring tree patterns [38], i.e., finding patterns within the DOM trees of multiple pages from the same domain and use dissimilarity information to extract content from webpages.

We will continue to work on clustering websites into useful clusters of genres and find algorithms for interpolating settings for websites that do not clearly fall into one cluster

Finally, we would like to use machine learning and natural language processing to learn user's browsing habits and workflow from work done in other applications and applying that knowledge gained towards content extraction.

7. 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. We have presented the changes made to our proxy through user interface and performance improvements. We have found that while the results that our original proxy produced were quite accurate, they had to be tweaked by hand for various websites that differ drastically in content and form. We have started to address that problem by dynamically detecting the context of the website, both in terms of physical layout as well as content genre. Using this information and comparing this to previously known results that work well for certain genres of sites, we are able to select settings for our heuristics and achieve the same results automatically that previously required a human administrator. We have also further improved our previously published results by supporting a wider variety of webpages through the support of CSS as well as Javascript-enabled websites. The Crunch framework provides the basis for additional research in context extraction and accessibility.

8. ACKNOWLEDGMENTS

The Programming Systems Laboratory is funded in part by National Science Foundation grants CNS-0426623, CCR-0203876 and EIA-0202063. Part of the work reported in this paper emanated from research from the Columbia IDS lab, which has been supported by grants form NSF and HS ARPA.

In addition, we would like to extend a special thanks to Peter Grimm, David Neistadt, Ke Wang, Janak Parekh and Phil Gross for their help towards this project.

9. 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 International Conference on Spoken Language Processing (ICSLP) 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] Suhit Gupta, Gail Kaiser, David Neistadt, Peter Grimm, "DOM-based Content Extraction of HTML Documents", 12th International World Wide Web Conference, May 2003.

[17] Suhit Gupta; Gail E Kaiser, Peter Grimm, Michael F Chiang, Justin Starren, "Automating Content Extraction of HTML Documents" Submitted to the World Wide Web Journal, January 2004

[18] Suhit Gupta, Gail Kaiser, "CRUNCH - Web-based Collaboration for Persons with Disabilities", W3C Web Accessibility Initiative, Teleconference on Making Collaboration Technologies Accessible for Persons with Disabilities, Apr 2003.

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

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

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

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

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

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

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

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

[27] Bauer, T. and Leake D., WordSieve: A Method for Real-Time Context Extraction. Modeling and Using Context: Proceedings of the Third International and Interdisciplinary Conference, Context 2001, Springer-Verlag, 2001.

[28] Gerard Salton. Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Inc., 1989.

[29] E. Stamatatos, N. Fakotakis, and G. Kokkinakis. Text genre detection using common word frequencies. International Conference on computational Linguistics, 2000.

[30] Brett Kessler, Geoffrey Nunberg, and Hinrich Schutze. Automatic detection of text genre. ACL/EACL, 1997.

[31] Shlomo Argamon, Moshe Koppel, and Galit Avneri. Routing documents according to style. International Workshop on Innovative Information Systems, 1998.

[32] Dmitri Roussinov, Kevin Crosswell, Mike Nilan, Barbara Kwasnik, Jin Cai, and Xiaoyong Liu. Genre based navigation of the web. International Conference on System Sciences, 2001.

[33] J. Karlgren. Stylistic experiments in information retrieval. Natural Language Information Retrieval. Kluwer, 1999.

[34] J. Karlgren, Ivan Bretan, Johan Dewe, Anders Hallberg, and Niklas Wolkert. Iterative information retrieval using fast clustering and usage-specific genres. *DELOS workshop on User Interfaces in Digital Libraries*, pages 85–92, Stockholm, Sweden, 1998.

[35] http://www.apache.org/~andyc/neko/doc

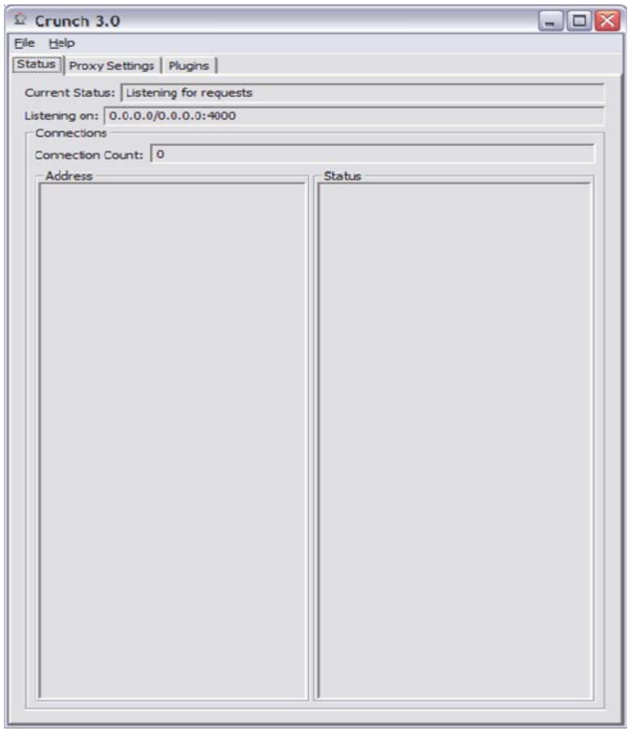
[36] A. Rauber and A. Muller-Kogler. Integrating automatic genre analysis into digital libraries. Joint Conf on Digital Libraries, 2001.

[37] Welsh, M. "The Staged Event-Driven Architecture for Highly-Concurrent Server Applications" Ph.D. Qualifying Examination Proposal, UC Berkeley, December 2000. http://www.cs.berkeley.edu/˜mdw/papers/quals-seda.pdf.

[38] Andrew Hogue, "Tree Pattern Inference and Matching for Wrapper Induction on the World Wide Web", Master's Thesis, Massachusetts Institute of Technology

[39] James Robertson, "So, what is a content management system?" KM Column, June 2003

10. APPENDIX

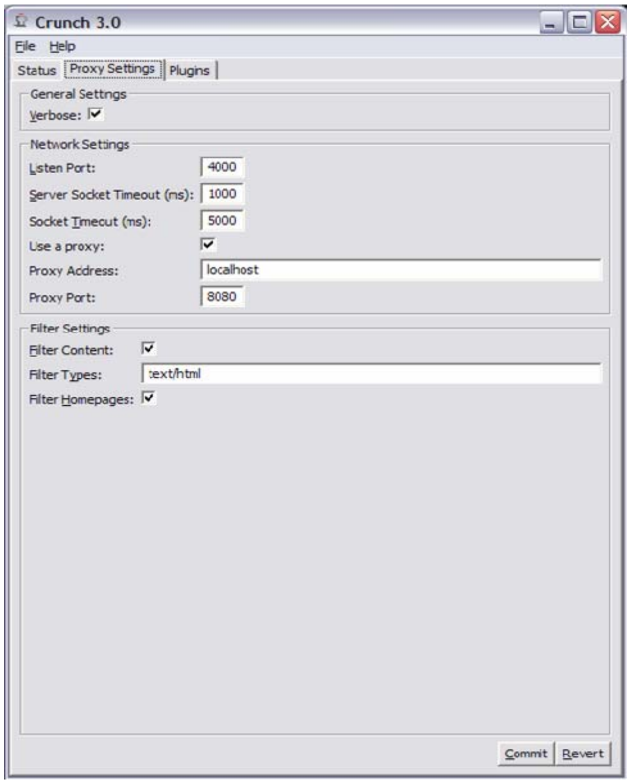


Current Status: Listening for requests

Listening on: 0.0.0.0/0.0.0.0:4000

Connection Count: 0

Figure 11 - Crunch 3.0 Status Window



Verbose: ✓

Listen Port: 4000

Server Socket Timeout (ms): 1000

Socket Timeout (ms): 5000

Use a proxy: ✓

Proxy Address: localhost

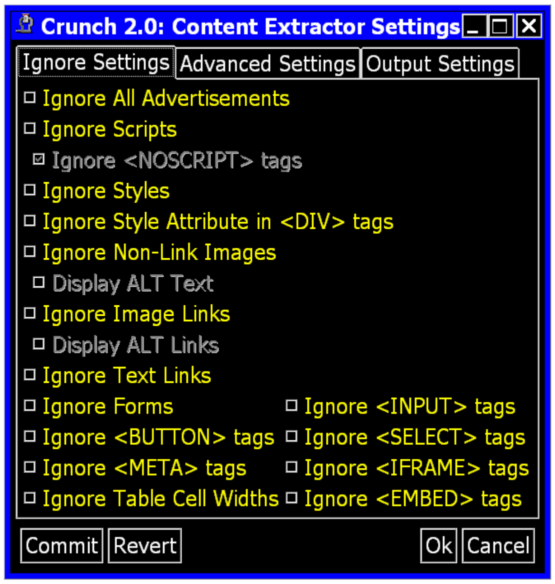
Proxy Port: 8080

Filter Content: ✓

Filter Types: text/html

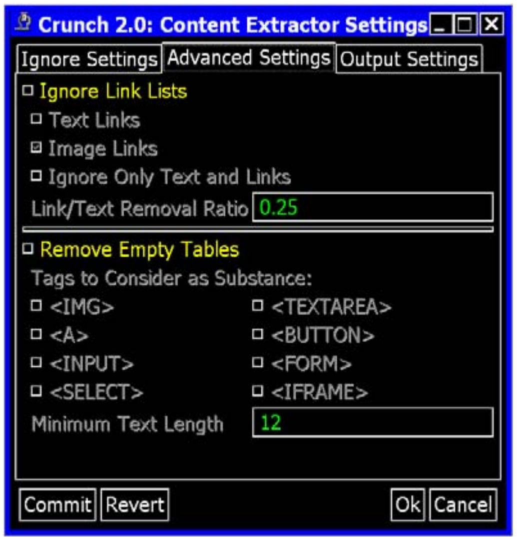
Filter Homepages: ✓

Figure 12 - Crunch 3.0 Settings Window



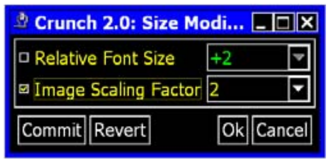
☐ Ignore All Advertisements
☐ Ignore Scripts
☒ Ignore <NOSCRIPT> tags
☐ Ignore Styles
☐ Ignore Style Attribute in <DIV> tags
☐ Ignore Non-Link Images
☐ Display ALT Text
☐ Ignore Image Links
☐ Display ALT Links
☐ Ignore Text Links
☐ Ignore Forms
☐ Ignore <INPUT> tags
☐ Ignore <BUTTON> tags
☐ Ignore <SELECT> tags
☐ Ignore <META> tags
☐ Ignore <IFRAME> tags
☐ Ignore Table Cell Widths
☐ Ignore <EMBED> tags

Figure 13 - Crunch 3.0 in contrast



☐ Ignore Link Lists
☐ Text Links
☒ Image Links
☐ Ignore Only Text and Links
Link/Text Removal Ratio 0.25
☐ Remove Empty Tables
Tags to Consider as Substance:
☐ <IMG>
☐ <TEXTAREA>
☐ <A>
☐ <BUTTON>
☐ <INPUT>
☐ <FORM>
☐ <SELECT>
☐ <IFRAME>
Minimum Text Length 12

Figure 14 - Crunch 3.0 settings in contrast



☐ Relative Font Size +2
☒ Image Scaling Factor 2

Figure 15 - Crunch 3.0 settings