Automatic Web Page Concatenation

6.UAP Final Paper

by Matthew Webber, 5/2005

Professor Rob Miller, Project Supervisor

Table of Contents

Introduction

Despite the many applications and resources designed to make retrieving information from the Internet easier, some tasks remain time-consuming and tedious for users. Consider the following three possible user scenarios:

Scenario 1: A user is using a web search tool to look for an image, and wants to quickly scan through as many images as possible. She wants to compare and contrast images that may have many search results between them. To do this, she may have to look at many different pages, possibly opening them in different windows. The image-heavy pages may take a long time to load, and switching between them may be difficult.

Scenario 2: A user is using a search utility to find products on a web site. After performing a search, he notices certain results are labeled "on sale". He wants to browse through the labeled results, but the site's search does not match the text of the "on sale" labels. He can use his browser's find utility to find the label on each page of results, but he must do this on each page of results one by one.

Scenario 3: A user has access to a news article that spans three web pages. She wants to print the article or save it to her computer, but the site she is using offers no "print friendly" version of the page. To perform either task, she must open each web page and print/save it separately.

In all of these scenarios, individual computer users are experiencing the same problem: they want to perform an action (view, search, save or print) on many different web pages at once. To do so, they must navigate to, load, and interact with each page separately. This process can be very repetitive and tedious, especially when large numbers of web pages are involved. In this paper, such a set of related web pages that a user might wish to combine will be referred to as a **Content Sequence**.

If the user had some way of viewing all content from a Content Sequence combined into a single page, actions could be performed on all data at once. The problem this project aims to solve is this: How can we empower computer users to combine Content Sequences from arbitrary web sites?

In this project, we will focus on a target audience of individual computer users. In general, these users can't write code, and don't understand the code of the web pages they use. They use many different web interfaces to retrieve information, and the sites they use change over time. A good solution to the problem will let users view pages of reorganized content when they want, and view other pages unchanged, without unnecessary distractions.

Related Work and Software Used

There have been many programs written that modify web content for end users. On a very basic level, HTML frames let users have a foreign sidebar beside web content. The web site that a user is viewing has little or no control over content that is displayed beside it. Some search engines display search results in a frame below their own content; Google still does something similar to this when it displays cached web pages.

More substantial page modification is possible, as well. Microsoft Smart Tags[4] automatically modify text, replacing certain words with links to relevant web pages. Ad blocking software lets users view a modified version of a page that has been stripped of certain content. Some tools go as far as to let end users develop their own scripts to modify web pages. Two extensions to the Firefox browser, Chickenfoot[1] and Greasemonkey, allow users to write scripts that change the way web pages behave. Chickenfoot also lets users develop code in a modified JavaScript language, and then insert them into a web page.

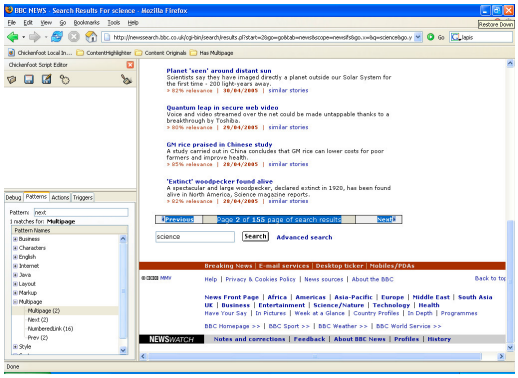

Figure 1: Chickenfoot running in the Firefox browser sidebar.

I used Chickenfoot for the bulk of the coding for this project for several reasons. First, Chickenfoot provides a convenient sidebar where code can be developed and executed (figure 1). The ability to write scripts alongside the target page, and to run the script at any time during the writing gave a considerable boost to speed at which I could develop scripts. Also, because the primary developer of Chickenfoot, Michael Bolin, is a member of MIT's User Interface Design Group, I was able to discuss my program and request small features and bug fixes easily.

Another advantage of Chickenfoot is its close relationship with the lightweight text manipulation tool Lapis[2]. Lapis provides Chickenfoot with the ability to recognize patterns in text and HTML such as links, numbers, HTML tags, and combinations thereof. That I could abstract away a layer of code and request the "first link containing number" from a page was very helpful for this project.

Algorithm Overview

In the following sections, I will describe the algorithm I created in this project to allow users to combine content in a Content Sequence. The algorithm has two parts. The first, which is run on all web pages a user views, focuses on detecting whether or not a web page is part of a Content Sequence. If a web page is part of a Content Sequence, the user will have the option to combine the content into a single page. If this option is chosen, the second part of the algorithm is executed. The second part isolates and extracts the content from each page, then combines it on one new page.

Detecting Content: Finding a Navigation Section

A program that concatenates web content may help some users on some web sites, but users probably will not want it making modifications to all web sites. Before we develop an algorithm that effectively combines web pages, we need to be able to tell, given an arbitrary page, whether or not it is part of a Content Sequence. We address this problem by looking for a feature common to almost every body of

content spanning multiple pages- a group of linked text or images serving as a table of contents. We will call this feature a page's **Navigation Section.**

Figure 2: Several different types of Navigation Sections.

Numbered Links

One feature common to the Navigation Sections on many web pages is a sequence of numbered links. These links can take many different forms (figure 2). They can be plain numbers, numbers with associated images, or numerical ranges. Since each link corresponds to one page in the content sequence, the numbered links are very useful for locating a number of pages in the content sequence. Unfortunately, a complex web page with numbered links could have many other words and numbers linked to web pages. The content detection algorithm I have developed, which is described a little later in this paper, is designed to separate Navigation Section numbered links from others.

Next and Previous Links

Another feature that appears in many Navigation Sections is a link or links for the previous and next page in the Content Sequence. While numbered links are usually dynamically generated and different on every page, Next and Previous links are more constant- on sites that use them, they appear exactly once per Navigation Section, in the same location, and on almost every result page (the first page in a Content Sequence does not have a previous link; the last page does not have a next link).

This constancy has advantages and disadvantages. The good part is that Next and Previous links are a common feature that can be used to identify Navigation Sections. The bad part is that since the rendered component of the next/previous links is not dynamically generated, a web site designer may easily replace them with images or form components. In cases where these components are not simple linked text, they become very hard for an algorithm to locate.

Navigation Section Finding Algorithm

This algorithm is meant to find all Navigation Sections on a single page.

1. Make a list of all links that contain a number or the strings "prev" or "next". These links may also contain images or letters. Start at the beginning of the list.
2. Iterate through all remaining links in the list, in the order that they appear in the HTML code.
3. Three types of links mark the start of a sequence: A link containing "prev", "1", or "2". Once one of these links is found, a sequence has begun. Subsequent links should be dealt with as follows:

   a. The expected next numbered link in a sequence should be added to the sequence. "3" is expected after "2", "7" after "6", and so on. "1" is expected after "prev".
   b. A number one greater than the expected next numbered link should also be added. ("5" after "3", etc.) This is because Navigation Sections may have a number corresponding to every page in a Content Sequence, but not make the number for the current page a link.
   c. If the next link contains "next", the "next" link is the end of the sequence. Store the sequence and go to step 2.
   d. If the next link does not match the conditions in a, b, or c, the sequence has ended. Store the sequence and go to step 2.
   e. If there are no more links in the list store the current sequence and continue to step 4.
4. After all links have been examined, find the longest stored sequence. Call its length L.
5. The Navigation Sections of the page are all sequences of length L.

Properties of the Navigation Section

This is the algorithm I've developed to find numbered links and Next/Previous links. The sequences it returns- consisting of links that are numbered, Next, or Previous- are the algorithm's best guess as to where a page's Navigation Section is. The performance of this algorithm is discussed in a later section of this paper.

Later use of the Navigation Section-finding algorithm will be based on two assumptions about Navigation Sections, which I will state explicitly here. First, Navigation Sections may be different from page to page, listing different numbered links or missing a Next link, but they will be the same within a page. Second, every page in a Content Sequence has a Navigation Section of some kind. This will be critical for my algorithm, as the way page content is located will depend on the location of the Navigation Section.

Content and Concatenation

In order to correctly combine a sequence of related web pages into one page, we need to be able to separate the material on the page into two categories: Content and Decoration.

Content is the material that the user cares about. Although it may be formatted in a repetitive way, it is unique on each page, and it is what motivates a user to click through multiple pages rather than staying on the first. In web search result pages, content is the links to pages that match the query, their descriptions, and related information. In a news article, the content is the title and paragraphs of the article. In a product search, it is the returned products, and their associated descriptions, images and prices.

Decoration is everything else- the things that give the pages context, structure, and supply the user with certain options. Decoration includes a page's title, header, sidebar, and footer. It includes links back to the main page, copyright information, and most advertising[1].

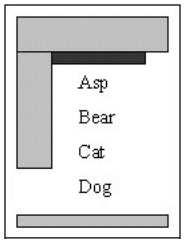

Figure 3: A simplified abstraction of a typical web page. The page content is a list of animals; decoration is in gray and the Navigation Section is in black.

To produce a single page that presents the content of every page in a Content Sequence to a user in a smooth and natural way, we will copy content from every page but the first. We will then insert it into the first page where the first page's content ends. Clearly, this course of action relies heavily on the ability to distinguish between content and decoration.

If decoration from pages is misclassified as content, headers and sidebars could be added to the synthesized page many times, which would add unnatural dividers between content and be visually distracting and confusing (figure 4). More seriously, if any part of the content is misclassified as decoration, that content would only be copied from the first page in the sequence.

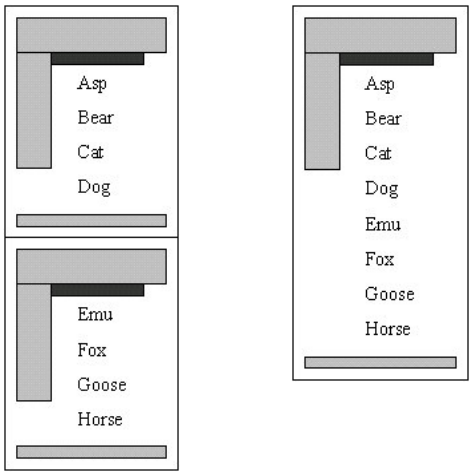

Figure 4: On the left, a naïve strategy for concatenating web pages. Placing the body of one page back to back with the body of another creates an unnatural division in the middle of the page. On the right, drawing only content from the second page yields a smoother concatenation.

[1] Advertisements that appear in the middle of content are rare, but do occur on some news web sites and search engines. Since they usually differ from page to page, for the purpose of this paper they will be considered content; how to best remove such advertisements is the job of an ad-blocking program and outside the scope of this project.

Consider, for example, a product search that returns a chart of products names, prices, pictures, and descriptions. If the left-most column (product names) was mistakenly classified as decoration, the final chart would only have product names for products that were originally listed on the first result page. All other products would be missing product names, and their columns might be shifted to the left, throwing off the structure of every column of the chart.

Missing content- cells in a table, a paragraph from an article, results from a search- is a serious mistake. Given a choice, we would much rather misclassify decoration as content than the other way around. For this reason, we will design algorithms that are biased toward content; they only classify something as decoration when they are sure it is decoration. Sometimes, these algorithms will mark decoration as content, but this is a necessary trade-off and will be kept to a minimum.

There are many algorithms that can be used to identify the content in a page. In the following section of this paper, I'll examine three, which I call the Spatial, Ancestor, and Range Algorithms. For each, I will explain the working of the algorithm, and how they make use of the Navigation Section described earlier in this paper. I will also go over the strengths and weaknesses of each algorithm. Finally, I will describe the algorithm I chose and explain why I chose it.

Spatial Algorithm

One striking observation that I made after examining dozens of sequences of web pages was that the content could always be closely contained inside a box on the screen. All the content was strictly below the header, above the footer, and to the right of any left-hand sidebars. Occasionally, text would wrap itself below the end of a right-hand sidebar, but such sidebars are usually small and not major features of a web page.

A Spatial algorithm, then, would look for bounds on the screen area the content occupied. It would search for a rectangle that would tightly wrap the page's content without excluding any of it. The Spatial algorithm has a very simple goal- it only needs to determine four numbers (the minimum and maximum values for both x and y that bound the rectangle).

The navigation sections that were described earlier in the paper are useful here. Since they usually appear immediately before or after the content in a page, they can be used to determine the minimum and maximum vertical coordinate of the content box. The browser Firefox provides functions that output the coordinates of any rendered HTML element's bounding box; these can be used to find the rendered position of the navigation sections, and later to find which parts of the page lie within the bounds of the content rectangle.

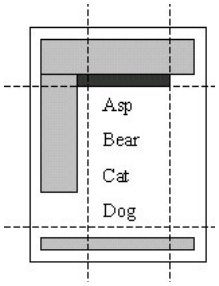

Figure 5: The Spatial algorithm determines four bounds for the content

The Spatial algorithm is not without its difficulties, though. Most web pages do not have features that lend themselves to giving vertical boundary lines (minimum and maximum x values). The algorithm is also very render-dependent. Due to the specifications of HTML, the same web page rendered in two different-sized windows often changes in height. As page height changes, the positions of the bounding boxes of HTML components change also. For this reason, a page whose content is correctly classified by the Spatial algorithm could cause errors if its window were resized, especially if it were resized to a very small or unexpected shape.

A more subtle drawback of the render-dependency of the Spatial algorithm is that it requires a rendering in the first place. Any browser extension that would fetch pages in the background and analyze and concatenate them without displaying them to the user first would have to go through many of the steps of rendering a page- loading image files to find their sizes, computing table cell widths, etc. The next two algorithms I describe, Ancestor and Range, are functions only of the page's source code and are much less restricted by rendering restrictions.

Ancestor Algorithm

Because HTML is (for the most part) a system of tags within tags, complex pages can be represented as a hierarchy or tree of HTML tags. In such a hierarchy, a table would be a <TABLE> node, which could have several table row <TR> nodes as children. Each of them would likely have table cell <TD> node children, inside which would be the words or further HTML elements that belong in each cell.

Nodes in the HTML hierarchy that correspond to elements rendered on the page take up a rectangle of space on the rendered page. With very few exceptions, the bounding rectangle of a parent node will completely contain the bounding rectangle of each of its child nodes. As discussed in the previous section, page content is generally bounded by a rectangle on the rendered page. Is there any relationship between the content rectangle and the HTML hierarchy rectangles?

As it turns out, there is. Every popular web search, product search, and news article on the web today is displayed on a page that is, in part or in full, put together dynamically. To design a complex system with

advertisements, cookies, sidebar links, and content that are managed by separate code (sometimes even separate servers) and automatically combined, a highly modular page design must be used. Content always closely corresponds to one or more sub-trees in the HTML hierarchy.

Just as with the Spatial algorithm, the Navigation Section is useful for the Ancestor algorithm. Since the

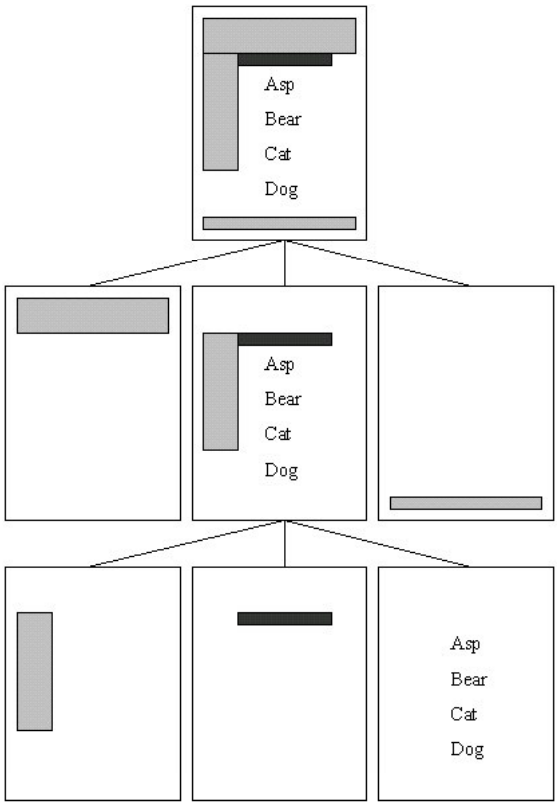

Figure 6: An HTML hierarchy.

Navigation Section is always physically close to the content and likely to be created by the same code as the content, it provides a good clue as to where the content of the page is. The content and navigation tags[2] are usually near each other in the HTML hierarchy; they commonly have the same parent node or grandparent node.

We can use this to our advantage. The position of a Navigation Section tag provides a good start point in the HTML hierarchy. If we iterate up the tree, examining higher and higher nodes, we are guaranteed to

[2] Here I'll refer to the HTML tag of the Navigation Section as the node deepest in the HTML hierarchy which fully encompasses the Navigation Section's content. The HTML tag of the page content is similarly defined. In some cases, these tags may contain page features not part of the Navigation Section or page content.

eventually find a node that includes the content of the page. All we need to do is find a way to stop on the right ancestor node and return it as our guess at the content tag.

This algorithm isn't perfect- the content tag will include the navigation section (whether or not we consider the Navigation Section part of the page's content), and may include some other small page decorations around the navigation section such as local headers "Search Results" or information about the results "displaying 11-20 of 243". But in general, there are usually one or more ancestor node that closely match the content and exclude all the biggest decorations on the page- headers, footers, and sidebars.

How, then, do we find the correct ancestor node? As we iterate up the HTML hierarchy from the navigation section, we can keep track of the number of characters of raw HTML code that each HTML node's tag encompasses. One important observation is that there is usually a single large jump in the amount of HTML code encompassed by these nodes. Nodes low in the HTML hierarchy, close to the Navigation Section contain little more than the Navigation Section. At some point going one parent node up reaches an ancestor of the content tag. An ancestor N of the page's content tag will usually have substantially more characters of HTML code than any child node of N that is not an ancestor of the content tag. An algorithm could select the node that follows after the largest increase in characters, or the largest percentage change in characters, or the first node that encompassed at least some percentage of the page's raw HTML code.

But each of these possible rubrics has its own weaknesses. Comments and scripts in HTML can take up little or no area on the page, but compromise a large portion of the code. Rendered area of an HTML node could be substituted for number of HTML characters, but this makes the algorithm dependent on rendering conditions like the size of the browser window.

Another problem with evaluating a node based on either rendered area or HTML characters is that the last page of a sequence sometimes contains a much smaller amount of content. Consider, for example, a web search that displays 10 results per page. A query that returns 21 results would have a good chance of breaking an algorithm that assumed the content of a page would be substantially bigger than the page's navigation section.

Range Algorithm

Although the Ancestor node algorithm only requires correctly guessing one value (the tree depth of the ancestor node) rather than four (the bounds of the rectangle), this simplicity limits it to only being able to select a single node. If some undesirable page feature is the direct child of the content node, no Ancestor algorithm rubric will be able to eliminate it without eliminating content. An algorithm that could pick a precise start and end point to the page content could eliminate any page decoration (unless the decoration appeared in the middle of the content, which is uncommon).

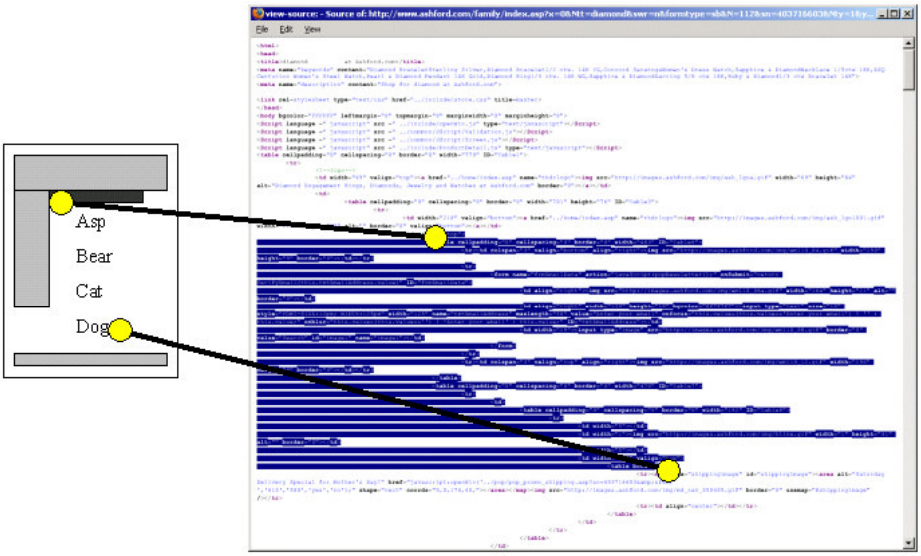

Figure 7: The Range algorithm finds a start and end point in the HTML source

But how to pick that start and end point? If there are two navigation sections, picking points before or after them is usually a pretty good approximation. But what if there is only one? It could be at the top or bottom of the page, so the side the content is on is not immediately obvious. The other endpoint of the content region will have to be the endpoint of the page, unless some page feature can be found to bound the content region.

Could an algorithm always select a certain HTML element for the endpoints of the range? Although there are certain types of tags that frequently enclose the content on a page, like <DIV> and <TBODY>, they usually appear many times in a page and enclose decorations, content, and other instances of their own tag. Short of hard-coding an algorithm to choose a certain content tag for each web site[3] it came across, there is no easy way to find the tags that start and end the content range.

Hybrid Algorithm

The three algorithms listed above each have their own advantages and disadvantages. To try to make use of the best parts of all of them, I developed a hybrid algorithm. The algorithm starts like the Ancestor algorithm, from the navigation section, and works its way upward. It examines the rendered size of each node's HTML element, and chooses the first one that intersects with the vertical halfway point of the page. I will refer to this halfway point as the page's equator.

[3] While this would be simple for some sites, like Google (first div tag in body), it would be tedious for others, like Yahoo! Shopping (form inside second div tag inside third div tag inside fifth div tag inside sixth div tag inside body)

Next, the hybrid algorithm uses the Spatial algorithm to eliminate any part of the page that lie (vertically) beyond the page's navigation sections. If a navigation section is above the equator, components above it are eliminated. If a navigation section lies below the equator, components below it are eliminated.

Finally, the algorithm returns a range from the first to the last point of the parts of the page still considered content. In effect, this fills in the gaps and replaces a possibly broken range with a single continuous one.

The performance of this hybrid algorithm is discussed in a later section.

User Interface

How does this all come together for a user who is performing a search or viewing and article?

When a user loads any web page, a script runs that checks for a Navigation Section in that page. If one or more is found, the script inserts linked text after each Navigation Section. The text of the link reads "Show All". When it is clicked, the script loads each of the Navigation Section's linked pages in a separate tab, determines their content, copies it, closes the tab, and inserts the content after the end of the content of the first page.

The Show All link has several beneficial attributes. Because the Show All link is inserted at the appropriate point in the HTML hierarchy as plain text, it automatically has the same font, color, and size as the text around it. This contributes to a very natural look for the link. The Show All link is small and unobtrusive, and the user only sees it when they are looking near the Navigation Section part of the page (presumably this is when they are thinking about the other pages in the sequence).

Most importantly, the Show All link leaves the decision of whether to substantially rearrange the web site's content in the hands of the user. If a section of a web page is misclassified as a Navigation Section, or a user only wants to see ten search results per page, they don't have to reconfigure their browser. They just have to avoid clicking "Show All" when browsing that site.

Unfortunately, in its current form the content is loaded, separated, and inserted one result page at a time. This is a long process that the user cannot easily stop halfway, and during the process the user can’t browse other pages in the same browser window. The script also leaves the modified page's URL unchanged, which could confuse some users who try to bookmark a concatenated page

Evaluation

During the development of my Navigation Section-finding algorithm, I frequently tested it on search results pages generated from a small set of web pages. In the following section I will describe my experiences with

finding the Navigation Sections on this training set of web sites, explain difficulties I encountered and problems I solved. Then, I will explain how my algorithm worked on a second set of sites, a set on which my algorithm was not tested on until it was completed. If my algorithm was overfitted to the training set, it will do poorly on the evaluation set.

I will count proper identification of all navigation section components present as a complete success, identification of at least one link per linked page in the Content Sequence as a half-success, and anything else as failure. Note that the Training Set and Evaluation Set sections only describe the performance of the Navigation Section-finding algorithm, not the content-finding algorithm.

Training Set

This training set included five web search pages, two product search pages, three news sites, and one government web site.

The web search pages were Google.com, Altavista.com, Yahoo.com, MSN.com, and Dogpile.com. Some had one navigation section per page, others had two. Some, like Google, had a linked image and linked text for every numbered link in the Navigation Section, while others only had a subtle linked sequence of text. All of the web search pages had Previous and Next links, as well as Numbered links. In my final round of testing all Navigation Sections were present and found correctly. For these pages, this was a 100% success rate.

The product search pages I trained on were Compusa.com and Ashford.com. Ashford had small left and right arrow images for its previous and next links, and these were not identified by my algorithm. The algorithm had to rely on the numbered links on this page, which worked fine on the first five search result pages, but failed on later pages because their numbered link sequences didn't start with 1 or 2 as expected. The algorithm successfully identified Previous, Next, and Numbered links on Compusa's web site. Product searches had a 50% success rate.

Of the remaining sites, the three news site searches fared well. Some had Numbered links and Previous/Next links, while others only had Previous/Next links. All were identified properly with no false positives. The government web site, Mass.gov, did not do as well. Previous and Next links were text links that displayed only angle brackets. Numbered links were in range form (1-10, 11-20, etc), and were not identified as an increasing sequence of links. These searches had a 75% success rate.

Evaluation Set

To further evaluate my algorithm, I tested it on fifteen web pages. I drew five pages from each of three categories: web search, product search, and news articles.

The five web search pages (Ask.com, Alltheweb.com, Hotbot.com, Webcrawler.com, Alexa.com) were the most successful, with a 70% success rate. Three were parsed successfully, and on one other the Previous and Next links were missed. On the other there was an error parsing the site.

The product sites (Amazon.com, Nordstrom.com, ParkAveElectronics.com, TheMallOfTheWorld.com, Target.com) were not as successful, with a 40% success rate. One was parsed correctly, two others only correctly identified numbered links, and no navigation features were found on the others. I found that failures were usually due to hard-to-identify features like stylized image-only buttons.

The news articles (Slashdot.org, Wired.com, Newsweek.com, Time.com, Reuters.com) also had a 40% success rate. This is pretty good, considering that the algorithm had never been run on this class of pages before (news site searches had been tested, but not news articles.) Most of the articles only included a link to the previous/next page in the article, so any non-recursive algorithm that tried to figure out the location of every page in a Content Sequence from the first page would have only found two pages worth of content.

False positives

To measure how often my Navigation Section-finding algorithm misclassified other page features, I ran it on several pages that had uncommonly difficult features.

First, I performed a Google search for the string "1 2 3". The first ten pages that were suggested all contained "1 2 3", or some close variant thereof. On each page, I ran the algorithm to identify numbered links. Only one of the ten pages, an article on CDC.gov, incorrectly identified two links as numbered links (they were actually footnotes). No other pages had numbered links.

Next, I examined the Google pages that listed search result pages for queries for "1 2 3", "prev", and "next". On these pages, there were a number of false positives. On the first, the algorithm identified 20 numbered links; there were actually only nine. On the second, the algorithm identified two links as Previous links when there was only one, and on the final page, the algorithm identified 10 Next links; there was only one.

Concatenated Pages

The previous sections only evaluated the algorithm that found the Navigation Section of a page. What about after the Navigation Section was successfully found, and other pages were loaded, analyzed, and inserted into the first page?

The pages that reached this stage did well. One first observation was that the content-preserving goal of the Hybrid algorithm was a success. Of the pages tested, every single page that reached the final concatenated state preserved all of the content from the sequence of web pages. Some of them did so by concatenating too much, in rare cases as much as the entire body of the page. But for the most part, the hybrid algorithm did an excellent job matching the content of the pages.

The concatenated pages also had a good look and feel to them. Some search result web pages, like Dogpile, numbered the returned search results, so there was a sense of continuity (figure 9). On other web pages that did not number or otherwise distinguish search results from each other, the navigation sections from the inserted pages provided context. (figure 8).

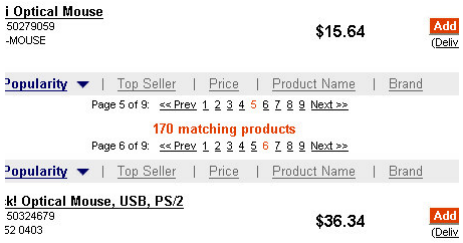

Figure 8: The Navigation Sections on this concatenated page give the user context- this area is between the content from page 5 and that from page 6.

Figure 9: The numbered results on this page are almost uninterrupted after concatenation.

Although the "look and feel" of a page is a very subjective measure, there are ways to quantify certain aspects of the process. When my concatenated pages were compared with naively concatenated pages (inserting each page's body in sequence), the difference in vertical page size was substantial. (There was no significant change in horizontal page size). On Dogpile.com, concatenating five search result pages reduced the page size by 15%. On Excite.com, five result pages were reduced by 17%. On sites with more search result pages, the reduction was even greater. On Compusa.com, nine result pages were reduced in size by 23%. Considering that this was done without eliminating any information the user is interested in, these savings are very significant.

Conclusions

In this project, I was able to create an algorithm that empowered users to combine together a wide variety of sequences of pages for browsing, saving, or searching. The Navigation Section-finding algorithm that I

created has been incorporated into the program Lapis, and the scripts that identify page content and concatenate web content have been added to the Chickenfoot code base. Although my code performed better on pages that I used while developing it, I believe it is general enough to work with many pages on the internet and flexible enough that it could be adapted to nearly any Content Sequence on the web.

Bibliography

1. Bolin, Michael. “End-User Programming for the Web”. 2005.
2. Robert C. Miller and Brad A. Myers. "Interactive Simultaneous Editing of Multiple Text Regions." *Proceedings of USENIX 2001 Annual Technical Conference*, Boston, MA, June 2001, pp 161-174.
3. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. "Automation and Customization of Rendered Web Pages." *Submitted to UIST 2005*.
4. Microsoft. "Smart Tags and Smart Documents." msdn.microsoft.com/office/understanding/smarttags/default.aspx