Harvey Mudd College Math Clinic 2002-2003

Three Methods for Improving Relevance Ordering for Web Search

Overture Services Inc.

Final Report

May 9, 2003

Participants

Erin N. Bodine	(Project Manager)
David F. Gleich	
Cathy M. Kurata	
Jordan A. Kwan	
Lesley A. Ward	(Faculty Advisor)
Daniel Fain	(Liaison)

Abstract

The 2002–2003 Overture clinic project evaluated and implemented three different methods for improving relevance ordering in web search. The three methods were bottom up micro information unit (MIU) analysis, top down MIU analysis, and proximity scoring. We ran these three methods on the top 200 web pages returned for each of 58 queries by an already existing algorithmic search engine. We used two metrics, precision and relevance ordering, to evaluate the results.

Precision deals with how relevant the web page is for a given query, while relevance ordering is how well-ordered the returned results are. We evaluated the precision of each method and of the algorithmic search engine by hand. For relevance ordering, we recruited other humans to compare pages and used their decisions to generate an ideal ranking for each query. The results of each of our methods and of the algorithmic search engine are then compared to this ideal ranking vector using Kendall's Tau.

Our bottom up MIU analysis method achieved the highest precision score of 0.78 out of 1.00. In addition, bottom up MIU analysis received the second highest correlation coefficient (or relevance ordering score) of 0.107 while the algorithmic search engine received the highest correlation coefficient of 0.121. Interestingly, our proximity scoring method received high relevance ordering scores when the algorithmic search engine received low relevance ordering scores.

Contents

Acknowledgments

We would like to thank Overture Services and Daniel Fain for giving us the opportunity to work on this project. Dan Fain has been instrumental in providing us with direction and advice. Additionally, we would like to thank Rosie Jones at Overture for her helpful contributions.

We would like to thank Professor Michael Raugh for his organization and direction of the Harvey Mudd Mathematics Clinic program. We also appreciate his administrative support that has helped our work on the project over the entire year.

We would like to thank Leslie Fletcher, a member of the Overture Summer Clinic team, for briefing us on their work. His continued contribution has helped move the project along.

We would also like to thank Barbara Schade for her invaluable aide in scheduling just about everything. Without her, our project would not have run nearly as smoothly as it did.

Lastly, we would also like to thank our advisor, Professor Lesley Ward. Her previous experience advising mathematics clinics has saved us substantial amounts of time. Also, her patient and positive attitude has greatly encouraged us.

Glossary

authority weight The weight a web page is given, depending on how good the information on this web page is.

base set The expanded root set which includes the original root set, as well as all the web pages pointed to by the root set, as well as some web pages that point in to the root set.

document object model (DOM) A tree based model of HTML where the tags are nodes in the tree and the text occurs at the leaves.

global algorithm A procedure that necessarily uses the entire web.

hub weight The weight of a web page that is dependent on the quality of the web pages it points to.

hyper-text markup language (HTML) The language of web pages. HTML describes a document using formatting tags to control the appearance of a page. HTML also describes hyper-links between web pages, the key feature linking the web together.

Kleinberg Algorithm Also known as Hyperlink-Induced Topic Search (HITS), this is an Internet search algorithm that takes into account the text-based searches and the hyperlinked environment of the Internet in order to return relevant search results to a given query.

micro informational unit (MIU) A segment of web page distinguished by its content and its display properties. A MIU contains information on only one topic. Li et al defined this in [LPHL]. See Chapter 5.1.1.

MIU analysis The process of dividing a web page into MIUs.

MIU breakup Synonymous with MIU analysis.

precision A measure of the accuracy of results returned from query search. If E is the set of web pages returned by the algorithm and H is the set of web pages judged relevant by a human then

$$precision = \frac{|E \cap H|}{|E|}$$

.

proximity scoring A procedure that ranks the relevance of a web page by the proximity of the words in a query to each other in the page.

query term A single word in a query.

root set The set of web pages, returned by a text-search engine, which contain the terms in the query. The Kleinberg algorithm expands this into the base set. See Section A.1.

sticky term A word or group of words in a multi-word query that may not have to appear near the rest of the query on the web page.

Chapter 1

Introduction

1.1 Web Searching

With the growth of the Internet, World Wide Web search engines have become both more important and more sophisticated. These search engines give order to the chaotic nature of the web. Without them, it would be difficult, if not impossible, to find any information on the web. For example, when a high school student researching the behavioral patterns of pygmy hippopotami enters the query "pygmy hippopotamus behavior" into the AltaVista search engine, the top results returned include a page from the University of Michigan Museum of Zoology archive with a section on pygmy hippopotamus behavior and a page from the America Zoo describing the pygmy hippopotamus and its behavioral patterns. This illustrates how search engines make the web particularly useful as an information source.

Search engines, and the algorithms behind them, have a hard problem to solve. They must infer the quality and importance of web pages without substantial human input. By itself, this is a difficult problem. But, as Chakrabarti et al [CJT] note, they face an even more difficult problem because they must also defend against individuals or groups of individuals attempting to compromise the validity of a search engine. Consequently, web search algorithms are an active area of research.

The problem Overture Services Inc. posed to our clinic team was to improve relevance ordering in web search. Our goal was to find a method that would return quantitatively better search results than existing algorithms. The specific problem statement for this clinic is in Chapter 2. This report summarizes a year's worth of research on this project.

1.2 Overture Services, Inc.

When Overture Services, Inc. began sponsoring clinic projects at Harvey Mudd College in 2002, they were not a search engine company, but rather a company that primarily worked with search engine companies as an auctioneer of words valuable to advertisers. Overture Services Inc. is a pay-for-performance Internet search provider. When Overture Services started up in 1997, search engines were not very organized. Some problems were that the Internet search results were randomly ordered, that many links returned by search engines were out of date or irrelevant to the query, and that there was a weak revenue model for advertisers. Overture Services addresses these problems by adding *sponsored links* to the search engine results of which include Yahoo, MSN, and Lycos. Overture auctions off key words to advertisers for these sponsored link spots. Specifically, when one of the key words appears in a query sent to a search engine, the link of the highest bidder for that word is presented at the top of the search results, as a sponsored link. The advertiser pays Overture Services each time someone clicks on the advertiser's link, hence the term 'pay-for-performance'. Also, in order to ensure relevant search results, Overture Services filters out advertisers whose businesses do not match up with the key word they bid on. Since then, Overture has also become a search engine company, having purchased the AltaVista and FastWeb search engines.

1.3 Overview of This Report

The remainder of this report contains a detailed explanation of our research, development and testing of three methods developed with the intent of improving the ranking of web search results. Chapter 2 discusses our problem statement and how our problem statement guided our research and implementation. Chapter 3 gives background information necessary for a detailed understanding of our re-ranking methods. It includes information on major theoretical approaches to web searching (global vs. local rankings), descriptions of micro information units (MIUs) and MIU analysis, and HTML document object model trees. Chapter 4 describes the big picture of our project. It discusses the approach we take and what metrics we use to produce our results. Chapter 5 describes the three methods we develop for re-ranking web search results we develop. This chapter contains a detailed explanation of how our three methods work. Chapter 6 explains how tests were conducted to determine how humans ranked results returned from a search engine. Chapter 7 gives a detailed description of the source code developed to implement our three methods.

Chapter 8 reviews the results we gathered from testing our three methods and includes a section on interesting observations about our data which, due to time constraints, were not explored. Chapter 9 describes our final conclusions about our three methods, including a discussion of the drawbacks of each of our methods and of the query set we used. Finally, we conclude with a discussion in Chapter 10 of extensions to our research we would have liked to explore.

We also include an annotated bibliography and a glossary of technical terms, as well as appendices containing our project timeline, related research, and selected pieces of our source code.

The full source code, all data collected, and tables containing the results of testing are contained on an accompanying CD.

1.4 The Accomplishments of One Year

In broad terms, we implemented three methods for re-ranking sets of web pages. The original web pages are found by an algorithmic search engine in response to a given query. Two of those methods parse web pages into micro information units (MIUs), and re-rank using on the content of each MIU. The idea is that a page would be ranked higher if all terms in the query appear in the same MIU, and lower if the query terms are scattered in different MIUs. The third method computes a proximity score for each page and re-ranks based on the magnitude of that score. The rankings produced by each of our methods was then compared to a standard ranking determined by human judgment and to the ranking of the algorithmic search engine.

We tested the re-ranking methods on a set of 30 queries. We used two metrics, relevance ordering and precision, to determine how good the re-rankings of each of our methods were. For some queries, some of our methods gave better rankings than the algorithmic search engines; for others the same or worse. Overall, we found that our methods of re-ranking did not show a statistically significant improvement over the algorithmic search engine's ranking. However, with a some suggested future improvements our methods may start showing a consistent advantage.

Chapter 2

Problem Statement

In this chapter we present original problem statement for out 2002-2003 Overture Clinic. We then present the revised problem statement that evolved out of discussions with our sponsor.

2.1 Original Problem Statement

At the beginning of the Fall 2002 semester, Overture Services gave us the problem statement:

> This project will build on previous Math Clinic projects on Web search, with the goal of improving techniques for topic dependent relevance ordering. Of the different approaches to ranking Web search results, some are static and global, using the Web as a whole to vote on the relative importance of pages, while others are more dynamic and local, ranking pages only within a search result cluster. In a purely static and global approach, if page A appears before page B when a user enters a search query Q, then A will appear before B for all other queries Q'. An example of a local technique is Kleinberg's HITS technique for link analysis; Brin and Page's PageRank is global. In a Summer 2002 Math Clinic, a specific approach was proposed for focusing link analysis, effectively making it more local. This 2002-2003 Math Clinic project will begin by evaluating that approach against some competing baseline approaches, then look at ways to blend local and global relevance scores, and compare local analysis within a topic cluster to local analysis within a search result set.

Since the beginning of the fall semester the 2002-2003 Overture Clinic Team has worked to refine the given problem into an area narrow enough to research implement in a one year period, but broad enough to touch on all of the various topics given in the original problem statement. The remainder of this chapter discusses the more definitive problem statement we designed.

2.2 Revised Problem Statement

The original problem statement stated in the previous section lays out three goals for the Overture Clinic Team.

1. Improve techniques for topic dependent relevance ordering.
2. Evaluate the Summer 2002 Overture Clinic's approach against competing baseline approaches.
3. Investigate ways to blend local and global relevance scoring.

The first major goal of improving techniques for topic dependent relevance ordering has been modified and made more specific. Instead of improving techniques, our main goal is to compare current techniques. In comparing existing methods we hope to determine if one technique has a better precision or relevance ordering than other techniques (See Section 4.2.1 and Section 4.2.2).

The second major goal of evaluating the Summer Clinic's approach against competing approaches has changed to a more basic question. Does the Summer Clinic's approach of MIU analysis done prior to link analysis seem reasonable? Thus, our goal of evaluating the Summer Clinic's approach became the more specific task of analyzing the Summer Clinic's approach and determining if it should be further pursued (see Section A.3.1 for full description of the Summer Clinic's approach).

The third major goal of investigating ways to blend local and global relevance scoring is rather vague and broad. In order to pursue other goals, this portion of the problem statement has been dropped. Instead, the idea of MIU [Micro Information Unit] analysis (see Section 5.1 for a full description of MIU Analysis) was further investigated, and then implemented and compared with another topic-dependent relevance ordering technique, namely proximity scoring.

Thus, our specific problem statement has these four main components:

1. Quantitatively test existing techniques for topic dependent relevance ordering in an attempt to determine if some techniques are "better" than others.
2. Analyze the Summer 2002 Overture Clinic's approach and determine whether the idea of MIU analysis done prior to link analysis should be further pursued.
3. Further investigate the idea of Micro Information Unit (MIU) analysis.
4. Implement one or more versions of MIU analysis and compare with an implementation of Proximity Scoring.

The term "better" is vague. For an explanation of the metric we will use to determine which technique is "better", see Section 4.2.1 and Section 4.2.2.

The subsequent chapters of this report contain our proposed solutions to the issues raised in this problem statement. We did this by concentrating our efforts on improving relevance ordering through our three methods: bottom up MIU analysis, top down miu analysis, and proximity scoring.

Chapter 3

Background

In this chapter, we describe some search algorithms which we will explore further in this paper. Some of these algorithms have been implemented by search engines, while others are recent ideas that could theoretically improve search results for a given query. We are primarily interested in improving the relevance ordering of the top few results for a given query. Our research is directed towards this end.

3.1 Web Searching

Web searching algorithms can be classified into two categories: *global* algorithms and *local* algorithms. Global search algorithms start with the entire web as the initial searching domain. Contrast this with a local searching algorithm, which starts with a subset of the web, often one related to a specific query, and uses the link structure of these pages to generate a more complete set to search on.

An example of a global algorithm is Google's PageRank algorithm developed by Brin and Page [BP]. Broadly speaking, Google takes the entire web and pre-ranks every single page using on a self-reinforcing link analysis process. That is to say, pages that have more links to them receive higher rankings, and those higher ranked pages give stronger scores to those pages that are linked to them. This process is done offline, before a query is entered. Once Google receives a query, it can just refer to the pre-ranked list and return the appropriate pages in order. In particular, if page A is ranked higher than page B in response to one query, it will also be ranked higher than page B for all other queries.

Many modifications have been made to PageRank, and researchers are still exploring ways to improve the algorithm. Ambiguous queries could perhaps be resolved by exploiting the context of the words that appear in the query; this is called topic sensitive search and is explored by Haveliwala [Hav] (see appendix A.4).

3.2 HTML DOM Analysis

The HTML Document Object Model (DOM) is a structured interface to manipulate HTML documents. [DOM]. HTML documents can be parsed according to the DOM structure [HWR].

This process is also known as HTML DOM Analysis. This is graphically represented by the HTML DOM tree (also known as the HTML tag tree), which is shown in Figure 3.1 [DOMTree]. HTML tag trees consist of a series of nodes. Each node in this tree has a tag name (e.g. <body>, <div>, <p>, etc.), content text, and its display attributes (e.g. color, font, size, etc.). For example, in Figure 3.1, the very top node has the tag, <body>, with no content text, and no display attribute. Each node in the tree also falls under the following categories: parent, child, or sibling. A parent node is an ancestor node of a node, while the child node is a descendant node of a parent node. If two nodes have the same parent node, then they are called siblings. In Figure 3.1, the bottom tags <b> and <i> are sibling nodes which are children nodes of the parent node, <p>.

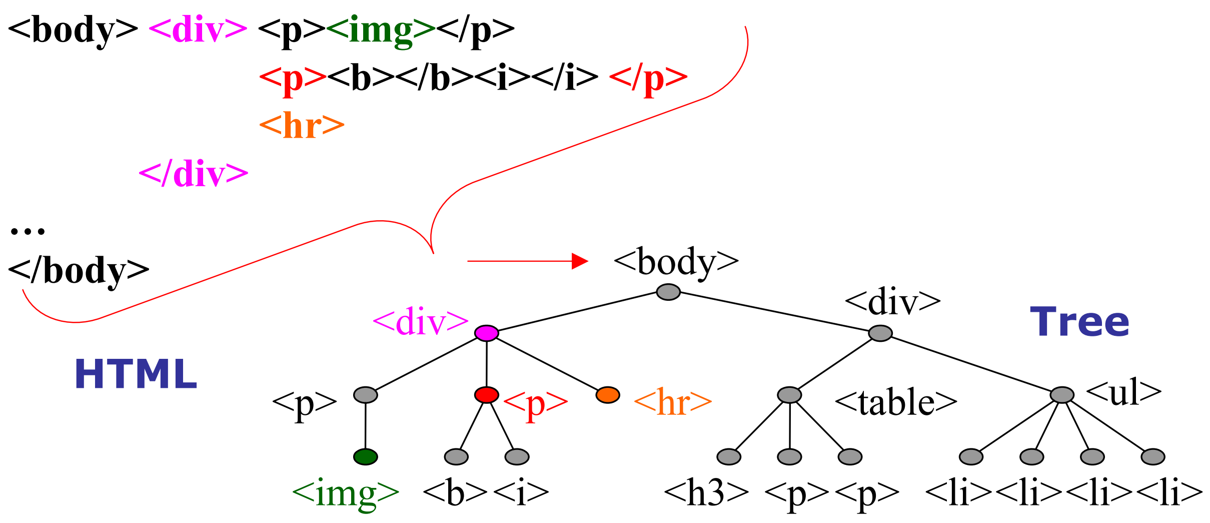


Figure 3.1: The HTML code in the upper left hand corner is parsed using the DOM structure, which then results in the DOM Tree, shown at lower right where each node is an HTML tag with no content text, and no display attributes.

3.3 Micro Information Units (MIUs)

Web pages often deal with more than one topic. This can lead to a search engine returning pages that are irrelevant to a given query, if the query terms do not occur close together on the web page but instead occur separately in parts of the web page dealing with different topics. Li et al [LPHL] proposed breaking up web pages into *micro information units (MIUs)*, that is, into blocks of text dealing with only one topic. This breakup process is called *MIU analysis*. In order to increase the precision of Internet search results, a web page can be run through a MIU analysis. This process exploits the HTML DOM structure described above. We implemented and tested two versions of MIU Analysis, and discuss more this in Chapter 5.

Chapter 4

Our Approach

This chapter outlines our approach to improving relevance ordering in web searches. We give a general overview with brief descriptions of the major components of the project, followed by an in depth explanation of how we re-ranked pages, and how those re-rankings were compared with a standard ranking based on human judgment.

4.1 Overview of Project

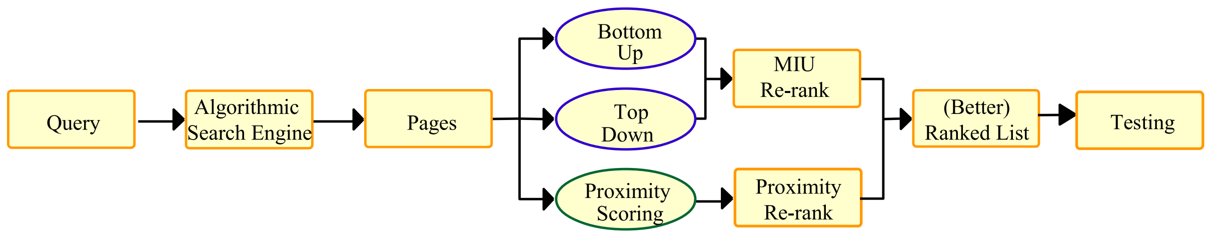


Figure 4.1: Pipeline diagram of an overview of our project. We take a query and run it through an algorithmic search engine. We then take the top 200 pages returned by the search engine and run them through our three methods: Top Down, Bottom Up, and Proximity Scoring. The first two methods return MIUs which are re-ranked, and the last returns whole pages which are re-ranked. These re-ranked results are then compared to a standard ranking based on human judgment.

A high level overview of our project is represented in graphical form in Figure 4.1. This graphical representation is called a *pipeline diagram*. It shows how we take a query and produce a ranking of pages for that query. We start by running a query through an algorithmic search engine. Due to time constraints, we used Google's SOAP protocol (see Section 7.3.1) to collect Google's top 200 pages for each of the 60 queries we used (see Section 6.1 for our queries). Although we used Google, our methods are generalizable to any algorithmic search engine. We then take these top 200 pages and run them through our three algorithms (bottom up MIU analysis, top down MIU analysis, and proximity scoring). The first two algorithms break each page up into one or more MIUs (see Section 5.1.1 for MIU analysis) and the last algorithm generates a numeric score based on the distance between sets of query words (see Section 5.2 for proximity scoring). Using the results from MIU analysis and proximity scoring, the originally collected 200 pages are re-ranked using two re-ranking methods we developed (see Section 4.1.2). The newly re-ranked lists are then compared to a ranking based on human judgment, and to the original Google ranking.

4.1.1 MIU Analysis and Proximity Scoring

During the course of our project we developed three methods to aid in re-ranking sets of web pages in order to improve relevance ordering. These three methods are *bottom up MIU analysis*, *top down MIU analysis*, and *proximity scoring*.

Each of our three methods were highly influenced by prior work. Our bottom up MIU analysis method was a reimplementation of Li et al's MIU segmentation [LPHL]. Likewise, our proximity scoring algorithm was from Hawking and Thistlewaite [HT]. In contrast to these two methods, our top down MIU analysis was inspired by Chakrabarti et al [CJT]. Chakrabarti et al proposed a top down tag tree analysis algorithm to identify dissimilar portions of a web page. We used their top down analysis idea to create our own top down MIU analysis method.

4.1.1.1 Bottom Up MIU Analysis

Bottom up MIU analysis breaks a page up into MIUs by starting at the bottom of the page's HTML DOM tree (see Section 3.2) and analyzing the tree upward. As the algorithm works its way up the tree, for each set of children nodes (nodes that share the same parent node) the algorithm determines whether the children are sufficiently similar. If they are not then they remain unmerged. If they are, then they are considered sufficiently similar to be in the same MIU, and the children nodes are merged together. In bottom up MIU analysis the tree is compressed as the algorithm progresses. (See Section 5.1.2 for details.) Our bottom up MIU algorithm is based on work done by Li et al [LPHL].

4.1.1.2 Top Down MIU Analysis

Top down MIU analysis breaks a page up into MIUs by starting at the top of the page's HTML DOM tree (see Section 3.2) and analyzing the tree downward. As the algorithm works its way down the tree, at each parent node the algorithm determines whether the children of that parent are sufficiently similar to the document as a whole. If they are sufficiently similar then the algorithm continues searching down the tree. If they are not sufficiently similar, i.e. they are sufficiently different, then the algorithm stops and the each child node is a separate MIU. In top down MIU analysis the tree is expanded as the algorithm progresses down the tree. (See Section 5.1.3 for details.) Our top down MIU algorithm is inspired by work done by Chakrabarti et al [CJT].

4.1.1.3 Proximity Scoring

Proximity scoring examines the distance between words. For a given query, proximity scoring computes a score for a passage of text (on a web page) based on how many words lie between the words in the query. Since there can be several terms in a query, our proximity scoring method cannot just compute the distance between *two* words in a query. To deal with multiple (more than two) query terms, we implement a proximity scoring method using the idea of a span. A span is simply a unique string of words that contain all the query terms. Thus, instead of computing the distance between two query terms, we compute the number of words in a span (which contains all the query terms). (See Section 5.2 for details.)

4.1.2 How Re-Ranking Occurs

We use two methods for re-ranking in our approach. One of the methods re-ranks pages in terms of MIUs, the other re-ranks pages according to their proximity scores. Both re-ranking methods try to take advantage of the original algorithmic search engine rank. Section 4.1.2.1 discusses re-ranking via MIUs and Section 4.1.2.2 explains re-ranking via proximity scores.

In this section, n denotes the number of terms in a query q, and p denotes a page.

4.1.2.1 MIU Window Re-Ranking

Once MIU analysis has broken a page into MIUs, we can use those MIUs to re-rank pages with the intention of improving precision. Although Li et al [LPHL] propose a method to do this, we devised a new method for our project. We describe above an overview of our MIU analysis (Section 4.1.1), and provide an in depth explanation in Sections 5.1.2 and 5.1.3.

Li et al focus on the number of query terms in each MIU in their re-ranking method. They compute two scores for each page, $primaryScore(p)$ and $secondaryScore(p)$. Here $primaryScore(p)$ and $secondaryScore(p)$ are the maximum number of query terms in a single MIU and any two adjacent MIUs, respectively. Both of these values are between 0 and n. Li et al then compute sets $pageSet_n$, $pageSet_{n-1}$, $\ldots$, $pageSet_1$, $pageSet_0$ where $pageSet_n$ contains all pages with $primaryScore(p) = n$ and any pages with $secondaryScore(p) = n$. In general, $pageSet_i$ contains any pages with $secondaryScore(p) = i$.

Li et al then generate the final ranked list of pages by ranking pages in $pageSet_{i+1}$ above pages in $pageSet_i$ (that is, rank in descending order) and among each $pageSet_i$ based on the original algorithmic search engine rank. Within $pageSet_i$, p_k is ranked above p_l if and only if $algorithmicRank(p_k) > algorithmicRank(p_l)$. For an example, see Figure 4.2.

Our method focuses, instead, on the number of MIUs required to contain all the query terms. Rather than compute two scores for each page, we compute $MIUWindowSize(p)$ which is the size of the smallest set of MIUs that contain all n query terms in query q. The value $MIUWindowSize(p)$ ranges between one and ∞. Like Li et al, we group pages into sets based on $MIUWindowSize(p)$. In our case, $MIUWindowSet_i$ contains all the pages with $MIUWindowSize(p) = i$. This relationship is formally expressed in Equation 4.1.

$$MIUWindowSet_i = \{p \mid MIUWindowSize(p) = i\}. \tag{4.1}$$

If $n = 2$ and

$$\begin{aligned} pageSet_2 &= \{p_2, p_5, p_3\} \\ pageSet_1 &= \{p_1, p_4, p_6\} \end{aligned}$$

then the final ranked list of pages from best to worst is

$$p_2, p_3, p_5, p_1, p_4, p_6.$$

In this example, pages p_2 and p_5 contain one MIU that had all n terms, so these pages belong in $pageSet_2$. Page p_3 contains all n terms in two adjacent MIUs and also belongs in $pageSet_2$. Pages p_1, p_4, and p_6 had at most one term in any pair of adjacent MIUs, so they belong in $pageSet_1$.

Figure 4.2: An example of Li et al's MIU re-ranking method. Here n is the number of terms in the query. p_i is the ith page from the algorithmic search engine, that is, $algorithmicRank(p_i) = i$. Within each set, Li et al order the pages by their $algorithmicRank$. For example, from $pageSet_2$, they rank page p_2 above page p_5.

We generate the final ranked set of pages by ranking pages in $MIUWindowSet_i$ above pages in $MIUWindowSet_{i+1}$ (that is, rank in ascending order) and, like Li et al, rank among each $MIUWindowSet_i$ using the original algorithmic search engine rank. For an example of our method, see Figure 4.3.

We thought our method would more effectively re-rank pages query terms spread over more than two MIUs. The pages in Figure 4.2 and Figure 4.3 are the same, but they are re-ranked differently by the two methods. In ours, the final rank of p_1 is less than every other page because it did not contain all the query terms, whereas Li et al's algorithm ranks p_1 as the fourth result.

4.1.2.2 Proximity Score Binning

Ideally, a re-ranking algorithm would incorporate the proximity scores into an existing ranking calculation involving all the search engine factors such as algorithmic score. However, without access to the internal search algorithm calculations, such a method is not possible. Instead, we devised a proximity score binning approach that exploits the both our proximity scores and the original algorithmic search engine rank. For details on how we compute proximity scores, see Section 4.1.1 for an overview, and Section 5.2 for more detail. Let $prox(p)$ be the proximity score of page p.

We first sort the pages in order of descending proximity scores, and call this vector $proxranks$. Thus, if $i < j$, $prox(proxranks_i) > prox(proxranks_j)$. We then take this vector and split it into sets, or bins, of size b. The expression for bin_i is formally defined in Equation 4.2. Finally, we rank bin_i above bin_{i+1} (that is, rank the bins in ascending order) and, like Li et al, we rank within bins by the algorithmic search engine rank. For an example of our proximity scoring re-ranking method, see Figure 4.4.

If $n = 2$ and

$$\begin{aligned} MIUWindowSet_1 &= \{p_5, p_2\} \\ MIUWindowSet_2 &= \{p_3\} \\ MIUWindowSet_3 &= \{p_4, p_6\} \\ MIUWindowSet_\infty &= \{p_1\} \end{aligned}$$

then the final ranked list of pages from best to worst is

$$p_2, p_5, p_3, p_4, p_6, p_1.$$

In this example, pages p_2 and p_5 contain one MIU that had all n terms, so these pages belong into $MIUWindowSet_1$. Page p_3 contains all n terms in two adjacent MIUs and also belongs in $MIUWindowSet_1$. Pages p_4 and p_6 contain all the query terms in at least three MIUs and belong in $MIUWindowSet_3$. Finally, page p_1 does not contain all the query terms and belongs in $MIUWindowSet_\infty$. Note that this example is consistent with Figure 4.2 and shows the alternate ranking induced by our method.

Figure 4.3: An example of our MIU window re-ranking method. Here, n is the number of pages in a query, and p_i is the ith page from the algorithmic search engine, that is, $algorithmicRank(p_i) = i$. Note that we rank page p_3 below page p_5

$$bin_i = \bigcup_{j=1}^{b} proxranks_{ib+j} \tag{4.2}$$

4.2 Comparing Results

We use two metrics, *precision* and *relevance ordering*, to measure the accuracy of our re-ranking methods.

4.2.1 Metric 1: Precision

The *precision* metric measures what fraction of the web pages returned by Google and by each re-ranking method judged by humans to be relevant to the query. The formula for precision is

$$precision = \frac{|E \cap H|}{|E|}. \tag{4.3}$$

Here, E is the set of ranked results from our method, and H is the set of results deemed relevant by humans. In our tests we computed the precision of the top ten results returned by each method, for each of 58 queries.

If $b = 3$ and

$$\begin{array}{rclrcl} prox(p_5) & = & 11.2 & prox(p_2) & = & 5.0 \\ prox(p_3) & = & 10.7 & prox(p_6) & = & 4.8 \\ prox(p_4) & = & 8.4 & prox(p_{10}) & = & 4.7 \\ prox(p_1) & = & 6.9 & prox(p_9) & = & 3.2 \\ prox(p_8) & = & 6.5 & prox(p_7) & = & 1.3 \end{array}$$

then

$$\begin{array}{rcl} bin_1 & = & \{p_5, p_3, p_4\} \\ bin_2 & = & \{p_1, p_8, p_2\} \\ bin_3 & = & \{p_6, p_{10}, p_9\} \\ bin_4 & = & \{p_7\} \end{array}$$

and the final list of ranked pages from best to worse is

$$p_3, p_4, p_5, p_1, p_2, p_8, p_6, p_9, p_{10}, p_7.$$

Figure 4.4: An example of our proximity score re-ranking method. Here, p_i is the ith page from the algorithmic search engine, that is, $algorithmicRank(p_i) = i$.

4.2.2 Metric 2: Relevance Ordering

Relevance ordering measures how "well ordered" the results returned by Google and our three methods are. That is, the orderings from Google and our three methods are compared to an "ideal ranking" that is constructed by humans. We use *Kendall's Tau* to measure how well correlated the re-ranked results are to the ideal ranking. The following section describes Kendall's Tau. These numbers are then compared to the correlation computed for Google against the ideal ranking. In our tests we applied the relevance ordering metric to each of 30 queries.

4.2.3 Kendall's Tau

Kendall's Tau takes in orderings of two data sets X and Y and returns a correlation measure that is always between -1 and 1. A measure of 1 indicates that the data sets are perfectly correlated, i.e. if X is data for weight and Y is data for height then a measure of 1 means that weight always increases with increasing height. Similarly, a measure of -1 means that the two ordered data sets are perfectly negatively correlated.

To compute this, order one set of data (say X) and line up the corresponding data for Y accordingly. An example is shown in Table 4.1. Then compare each data point in Y with the ones below it. If a data point below is greater than the data point in question, it is said to be *concordant*. Similarly, if a data point below is less than the data point in question, it is said to be *discordant*. Let C and D be the numbers of *concordant* and *discordant* pairs, respectively, and let n be the size of the data sets. Then *Kendall's Tau* is computed as follows:

$$\tau = \frac{C - D}{\binom{n}{2}}. \tag{4.4}$$

Height (X)	*Weight (Y)*	*Concordant*	*Discordant*
60	100	9	0
63	105	8	0
65	120	6	1
67	117	6	0
68	170	4	1
70	150	4	0
72	225	0	3
73	215	0	2
74	210	1	0
76	211	0	0
total		*38*	*7*

Table 4.1: In this example, data for X is ordered, and the corresponding data for Y follows. To illustrate, the number 8 under "*Concordant*" represents the number of times the weight 105 is less than weights in the rows below it. Thus, the numbers under "*Concordant*" represent the number of *concordant* pairs for a given datum from Y, and the total is shown below. The *Discordant* column is similar, except it counts the number of data points in Y that are *greater* than the datum in the same row. Here, n is equal to 10, as there are 10 pairs of data. Hence there are $\binom{10}{2}$ or 45 pairs in all. Subtracting 7 from 38 and then dividing out by the total gives $\tau = .689$. This value of τ suggests that this data is strongly positively correlated.

Chapter 5

Overview of Our Three Methods

5.1 MIUs

5.1.1 Theory

In a 2002 paper [LPHL], Li et al define the concept of a micro information unit (MIU) and present an algorithm for bottom up MIU analysis. They define an MIU as a segment of a web page, distinguished by its content and display properties. Further, each MIU should be topically cohesive. More intuitively, an MIU is just a small piece of information within a larger web page. Figure 5.1 shows a sample page divided into MIUs.

Li et al intended MIUs as a search engine feature to resolve queries with low precision (see Section 4.2.1 for more detail on precision). The example Li et al used is the phrase "free download video." Entering this query into the Google search engine returns a series of highly authoritative, but not particularly relevant, pages to the search query. When they performed this search, the first ranked result returned was Real Inc.'s RealPlayer website. This page offers a free download of a video player (this is why Google returned it) but never discusses the real intent of the query – freely downloadable videos. In fact, the terms "free," "download," and "video" all occur in topically distinct areas of this page.

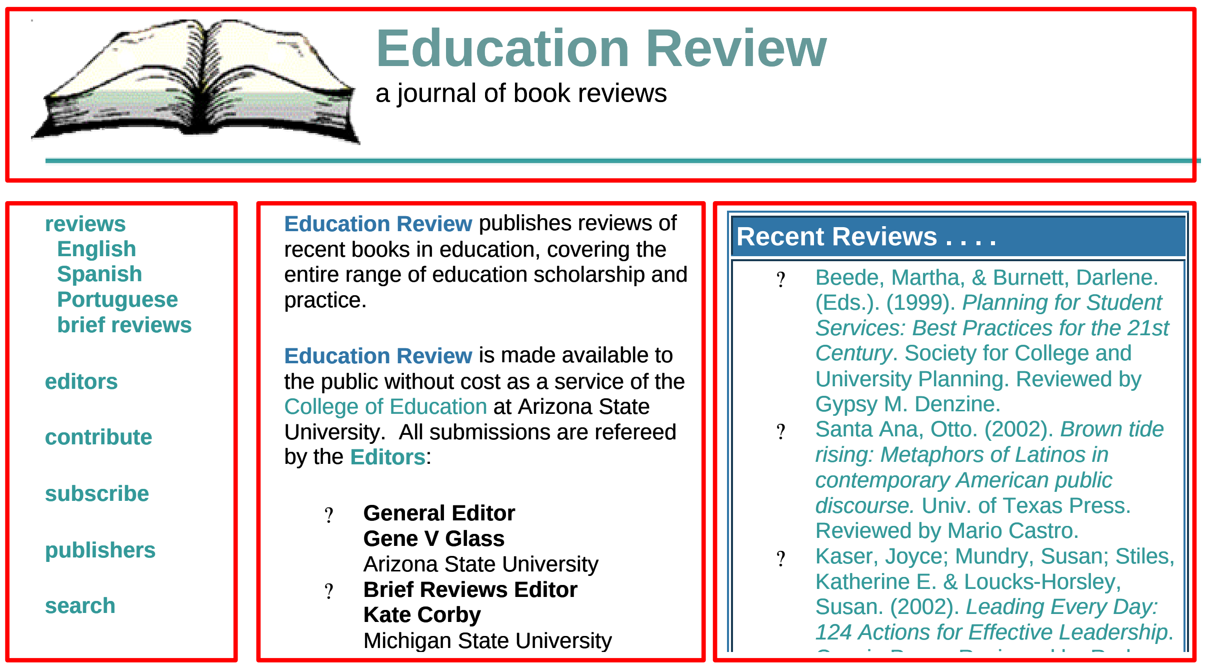
Thus, MIUs help resolve such multi-term queries. After the initial ranking, the search engine re-ranks the results and gives higher scores to those pages with MIUs that contain all of the query words. This strategy eliminates erroneous results like the previous situation.

The difficulty with MIUs, then, lies in finding the MIUs on a page. This is the process of *MIU analysis*. The next two sections describe the two methods of MIU analysis we used. In Section 10.2, we propose more complicated but untested algorithms for MIU analysis.

5.1.2 Bottom Up (following Li et al)

Our bottom up MIU analysis is a reimplementation of Li et al's algorithm [LPHL]. In this section, we present Li et al's algorithm from their paper. The small deviations from their algorithm are discussed in Section 7.1.4.

Li et al propose an algorithm for this process that operates on an HTML tag tree. The algorithm first parses the hyper-text markup language (HTML) into a DOM tree. See



Education Review

a journal of book reviews

reviews
English
Spanish
Portuguese
brief reviews

editors

contribute

subscribe

publishers

search

Education Review publishes reviews of recent books in education, covering the entire range of education scholarship and practice.

Education Review is made available to the public without cost as a service of the College of Education at Arizona State University. All submissions are refereed by the Editors:

? **General Editor**
Gene V Glass
Arizona State University
? **Brief Reviews Editor**
Kate Corby
Michigan State University

Recent Reviews . . . .

? Beede, Martha, & Burnett, Darlene. (Eds.). (1999). *Planning for Student Services: Best Practices for the 21st Century*. Society for College and University Planning. Reviewed by Gypsy M. Denzine.
? Santa Ana, Otto. (2002). *Brown tide rising: Metaphors of Latinos in contemporary American public discourse.* Univ. of Texas Press. Reviewed by Mario Castro.
? Kaser, Joyce; Mundry, Susan; Stiles, Katherine E. & Loucks-Horsley, Susan. (2002). *Leading Every Day: 124 Actions for Effective Leadership.*

Figure 5.1: This figure demonstrates one possible division of the web page `http://www.ed.asu.edu/edrev/` into MIUs. The MIUs are the boxes in the figure.

section 3.2 for more details on the DOM tree. The DOM tree is the tag tree used in the rest of the algorithm. Next, the algorithm performs stemming and stop list culling on the text in the DOM tree to simplify later textual processing.

In order to segment the page properly, Li et al define two cases for merging nodes.

1. Merge heading paragraphs with content paragraphs.

2. Merge adjacent text paragraphs.

Li et al define the following helper functions to use in identifying which nodes to merge.

- $length(A)$ = the number of words in node A.

- $tagRank(A)$ = the display emphasis of node A, that is, a value based on the HTML tag of A. The most highly weighted tags are those that specifically denote headings, such as <h1>, <h2>, etc. Next are formatting tags such as, <b>, <strong>, <i>. Finally, font tags such as <font>, <size>, <big>. All other tags have lower values than these three classes.

- $neighbor(A, B)$ = true if node A is the left sibling/neighbor of node B in the tag tree.

- $displaySimilarity(A, B)$ = the number of common display features in nodes A and B. Display features include fonts, sizes, and colors.

- $contentSimilarity(A, B)$ = the number of common words in nodes A and B.

Sections 5.1.2.1 and 5.1.2.2 expand upon how Li et al identify nodes to merge. Section 5.1.2.3 explains the entire MIU analysis algorithm. Finally, Section 5.1.2.4 contains a visual depiction of the algorithm running on a hypothetical tag tree.

5.1.2.1 Merging Header and Content Paragraphs

Li et al establish two conditions for two nodes to exist in a header-paragraph relationship. First, given a node A and a node B, A is a potential header to B if A has a greater *tagRank* than B, if the length of A is strictly less than the length of B, if A is B's left neighbor, and if A and B share at least one word. Equation (5.1) formalizes this condition.

$$\begin{aligned} &headerContent(A, B) = \\ &\quad (tagRank(A) \geq tagRank(B)) \wedge (length(A) < length(B)) \wedge \\ &\quad neighbor(A, B) \wedge (contentSimilarity(A, B) > 0) \end{aligned} \tag{5.1}$$

Intuitively, this condition looks for two adjacent nodes looking like a header and a paragraph that share at least one word. Typically, a header has a stronger visual emphasis than the following paragraph. Headers are also shorter than their following paragraphs.

However, this condition is insufficient to fully identify a header-paragraph relationship. In addition, there must exist another pair of nodes C and D, such that $headerContent(C, D)$ holds. Also, the display similarity between nodes A and C and the display similarity between nodes B and D must be greater than some constant, δ. Li et al experimentally determined $\delta = 3$. Equation (5.2) specifies this condition.

$$\begin{aligned} &HeaderAndParagraph(A, B) = \\ &\quad headerContent(A, B) \wedge \exists (C, D), headerContent(C, D) \wedge \\ &\quad (displaySimilarity(A, C) \geq \delta) \wedge (displaySimilarity(B, D) \geq \delta) \end{aligned} \tag{5.2}$$

This condition defines when two nodes have a header paragraph, content paragraph relationship. Note that if $HeaderAndParagraph(A, B)$ holds, then there must exist another pair (C, D) such that $HeaderAndParagraph(C, D)$ also holds. Thus, for all pairs (A, B) and (C, D) the algorithm merges nodes A and B, as well as all possible nodes C and D.

5.1.2.2 Merging Adjacent Text Paragraphs

Li et al define a simple condition that must hold to merge two adjacent text paragraphs. The only stipulation is that the nodes be adjacent and share at least some number of words, $\overline{\omega}$. Through experimentation, they concluded $\overline{\omega} = 2$ sufficiently captured adjacent node similarity. Since the textual comparison occurs after stemming and stop-list culling, this requirement makes sense. Equation (5.3) expresses this condition.

$$\begin{aligned} &ParagraphAndParagraph(A,B) = \\ &\quad neighbor(A,B) \wedge contentSimilarity(A,B) \geq \overline{\omega} \end{aligned} \tag{5.3}$$

5.1.2.3 Li et al's Algorithm

Li et al's overall algorithm first fully segments the page using the HTML DOM parse tree, then attempts to merge the bottom-most nodes in this tree and continually steps up through the tree seeking new nodes to merge. Consequently, we call this algorithm a *bottom up* approach.

In the algorithm, Li et al use a few variables relating to the DOM tree.

- $maxDepth$ is the maximum depth of the tree.
- $treeDepth$ is the depth where the algorithm is currently working. This value iterates from $maxDepth - 1$ to 0.
- $subtrees$ is the set of sub-trees with a root node at $treeDepth$.
- $subtree_i$ is the ith sub-tree with only leaf nodes, in the set $subtrees$.

See Figure 5.2 for the algorithm. Roughly, the important aspects of the algorithm are:

1. Merge header and content paragraphs.
2. Merge adjacent text paragraphs and restart the algorithm at step 1 if anything is merged.
3. Merge singleton children with their parents.

5.1.2.4 Example of Li et al's Algorithm

Figures 5.3 through 5.8 visually depict how Li et al's algorithm uses a tag tree to determine MIUs in a document. The figure captions contain the narration associated with our hypothetical example.

```
LiBottomUpMIUAnalysis($tree$)
for $treeDepth \leftarrow maxDepth - 1$ to 0
do $subtrees \leftarrow \{subtree_i \mid subtree_i$ is a sub-tree at level $treeDepth\}$
    while $|subtrees| > 0$
    do for each $subtree_i \in subtrees$
        do if $|subtree_i| = 2 \wedge$ headerContent($A, B$)
              then Merge($A, B$)
              else for each $(A, B)$, neighbor($A, B$)
                    do if headerContent($A, B$)$\wedge$
                          HeaderAndParagraph($A, B$)
                          then for all nodes $(C, D)$ such that
                                HeaderAndParagraph($A, B$)
                                do Merge($C, D$)
                                endfor
                                Merge($A, B$)

        endfor
        for each $subtree_i \in subtrees$
        do for each $(X, Y)$ such that neighbor($X, Y$)
            do if ParagraphAndParagraph($X, Y$)
                  then Merge($X, Y$)
                        goto line 4
            endfor
        endfor
        for each $subtree_i \in subtrees$
        do if $A$ has no $sibling$
              then Merge($A, parent(A)$)
        endfor
    endwhile
endfor
```

Figure 5.2: Li's algorithm

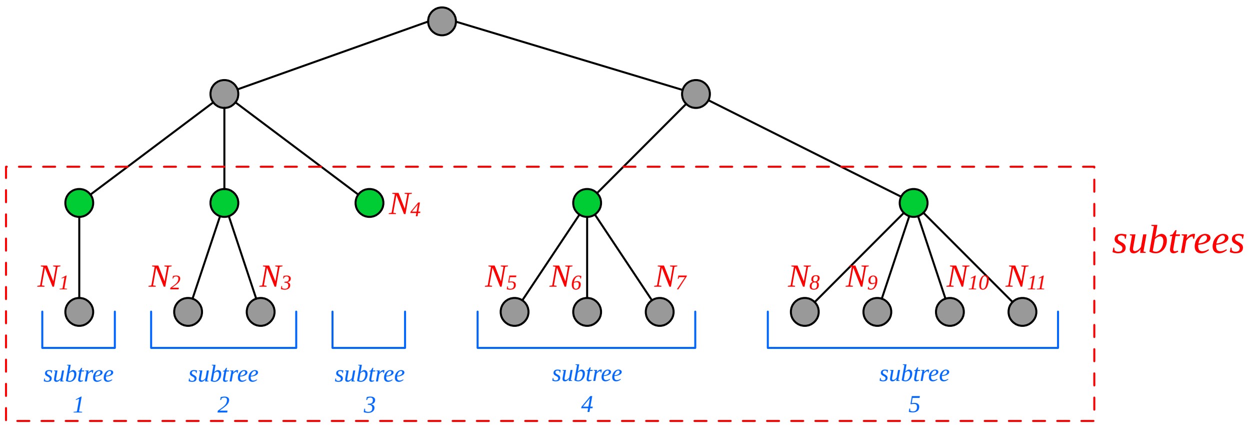


Figure 5.3: First figure in Li et al example. Given the tag tree depicted in this figure, the algorithm begins by looking at the bottom level of the tag tree. The set $subtrees$ consists of all the subtrees enclosed inside the dashed box, $subtree_1$ through $subtree_5$. The node labels, N_1 through N_{11} are used in the following diagrams to clarify ambiguities. The algorithm begins by examining $subtree_1$. It finds a singleton node, which the algorithm merges with its parent. This step completes processing of $subtree_1$. Moving to $subtree_2$, the algorithm first tests $headerContent(N_2, N_3)$, which returns true. Since $|subtree_2| = 2$, nodes N_2 and N_3 are merged.

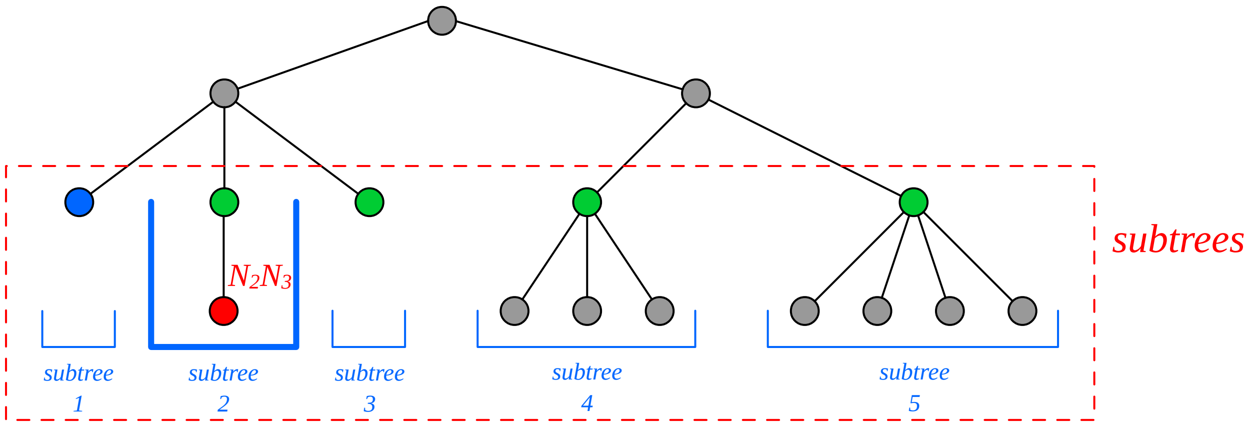


Figure 5.4: Second figure in Li et al example. Here, the algorithm reexamines $subtree_2$ to look for singleton nodes. It finds the new merged node N_2N_3 and merges this node with its parent. Now, the algorithm moves to $subtree_3$. This subtree has no children, so the algorithm ignores it and moves to examine $subtree_4$. First, it tries to merge nodes N_5 and N_6 using the $headerContent$ test. This fails. It next examines $headerContent(N_6, N_7)$, which succeeds.

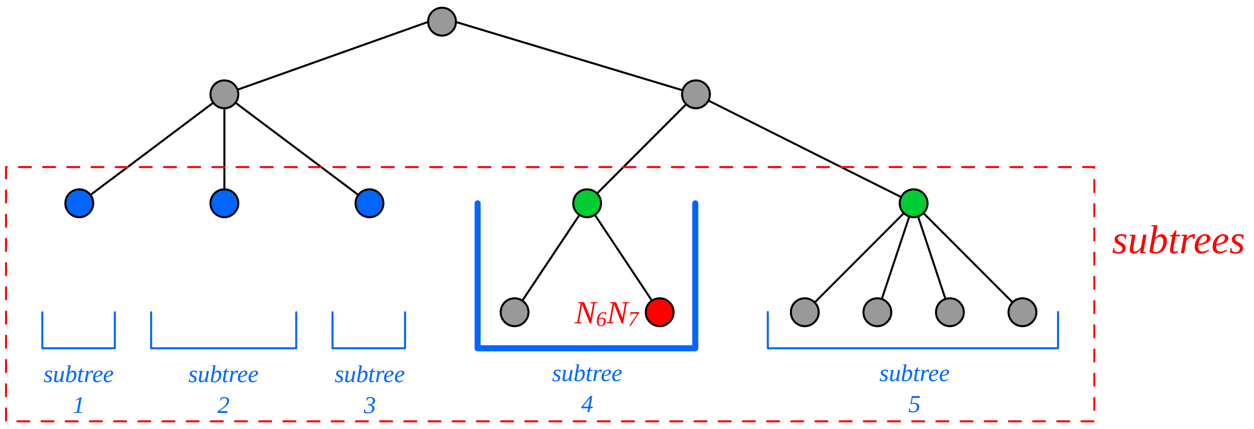


Figure 5.5: Third figure in Li et al example. After previously merging nodes N_6 and N_7, the algorithm restarts and attempts to find header-content relationships again. Now, $headerContent(N_5, N_6N_7)$ succeeds, and these nodes are merged into $N_5N_6N_7$, which is then merged with its parent.

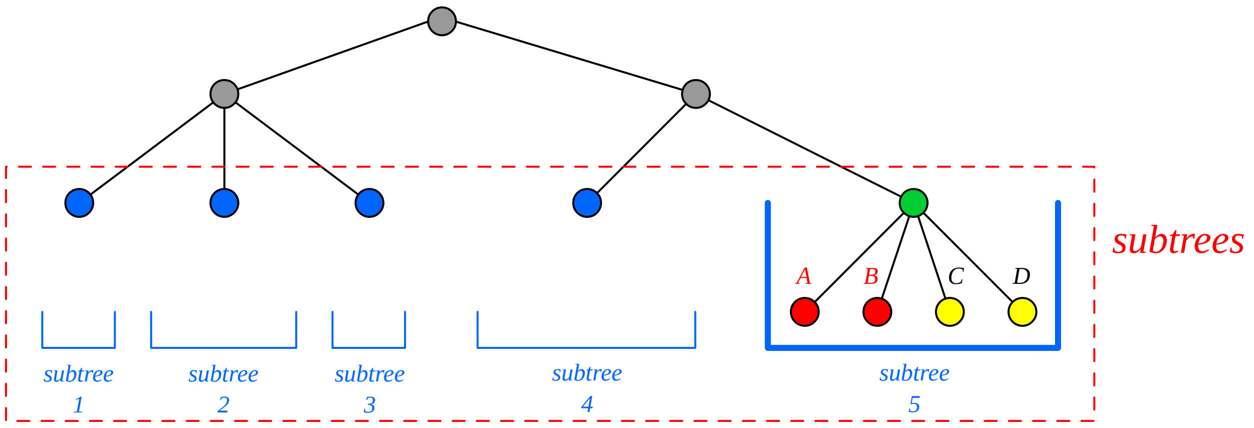


Figure 5.6: Fourth figure in Li et al example. When the algorithm considers $subtree_5$, it first finds that $headerContent(A = N_8, B = N_9)$ is true. It then looks for nodes C, D such that $HeaderAndParagraph(A, B)$ holds. The algorithm finds $C = N_{10}, D = N_{11}$, and merges these nodes. Since it does not find any other nodes C, D, it merges N_8 and N_9. Next, the algorithm attempts to merge the nodes N_8N_9 and $N_{10}N_{11}$ as adjacent text paragraphs, which fails. Since this is not a singleton subtree, the algorithm does no more merging on $subtree_5$. The algorithm now moves to the next level in the tag tree.

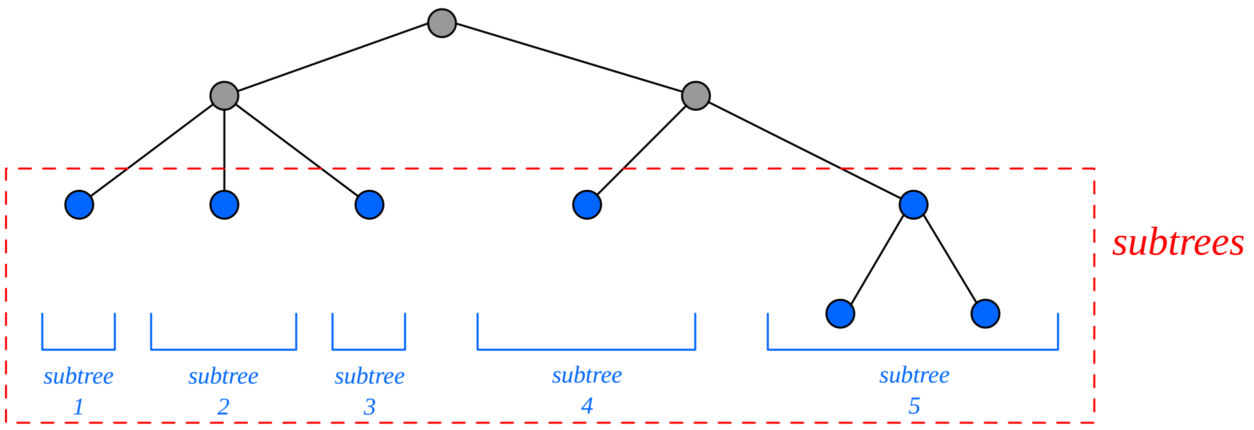


Figure 5.7: Fifth figure in Li et al example. In processing the higher levels of the tag tree, the algorithm does not merge any other nodes. This is the final merged tag tree.

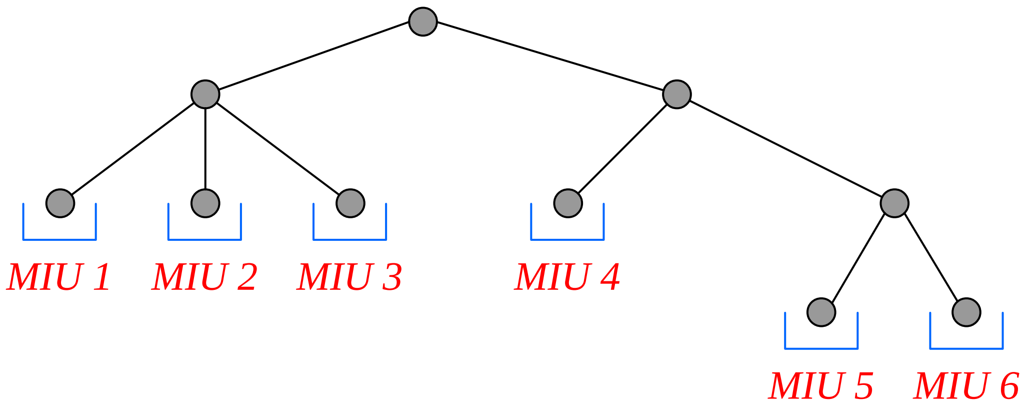


Figure 5.8: Sixth figure in Li et al example. Here, the algorithm has completed and returns each leaf node as an MIU.

5.1.3 Top Down (following Chakrabarti et al)

Our algorithm for top down MIU analysis is based on a paper by Chakrabarti et al [CJT]. In that paper, they introduce the idea of micro-hubs as an improvement in the Kleinberg algorithm. In many ways, Chakrabarti et al's micro-hubs are analogous to our MIUs. They intended their micro-hub analysis to identify portions of a web page that are too dissimilar from the rest of the root set in the Kleinberg algorithm. Presumably, these dissimilar portions are irrelevant links and might constitute a clique attack against the search algorithm.

Like Li et al [LPHL], Chakrabarti et al start micro-hub analysis using the HTML DOM tree. In contrast to Li et al, Chakrabarti et al start their micro-hub analysis at the top of the HTML DOM tree. We adapted their micro-hub analysis to devise the top down MIU analysis algorithm.

The fundamental idea behind our top down MIU analysis algorithm is that MIUs should not be too similar to the entire document. To implement this idea, we first parse the HTML of a page into an HTML DOM tree (see Section 3.2).

Figures 5.9 and 5.10 illustrate the algorithm. Starting from the root node A at the top of the tree, we create an MIU for each child node a of A. We then attempt to split the MIUs based on how similar the text of a is to the rest of the document d. That is, we first compute the *similarity measure* for a using Equation 5.4.

$$similarity(a,d) = \frac{\sum_t w_{t,b1} w_{t,b2}}{\sqrt{\sum_t w_{t,b1}^2 \sum_t w_{t,b2}^2}}. \tag{5.4}$$

In this equation, a is the child node in question, d is the document, and $w_{t,bj}$ is the weight of term t in text block j, which is just the frequency of term t. The *similarity measure* returns a value between 0 and 1, and is a quantitative measure of how similar a is to d. This value is then compared against a threshold value δ. If $similarity(a,d) < \delta$, then node a is considered dissimilar enough from the document, and so the child remains its own MIU. If $similarity(a,d) > \delta$, then a is too similar to the document and is split into its children, so the algorithm repeats this process on the child nodes of a. The program terminates when there are no more child nodes left to repeat the process.

5.2 Proximity Scoring

A few formulas for proximity scoring are reported in papers by Hawking and Thistlewaite [HT], Radev et al [RFQWG],and Clarke and [CC]. These formulas are very basic and intuitive; they capture the notion of proximity by measuring the distance between terms in the query, inverting the distance, and summing all the distances.

$$P_{(q,d)} = \sum_{i=1}^{j} \frac{1}{|\sigma_{(q,i)}|} \tag{5.5}$$

Here, q is the query, d is the document, j is the number of instances of the query words in the document, and $|\sigma_{(q,i)}|$ is the number of words, or the distance, between the ith occurrence of the query terms. To handle overlapping occurrences of the query words, an instance is only

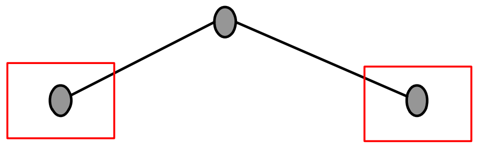


Figure 5.9: Top down first examines the child nodes just below the head node. The child nodes are compared in similarity to the head node using *similarity measure*. The top down algorithm then checks whether $similarity(a, d) < \delta$ for each node a.

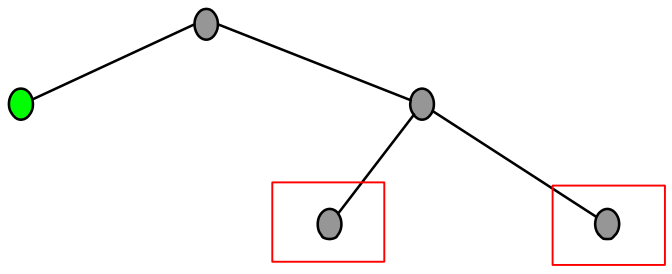


Figure 5.10: The first iteration completes when top down determines that $similarity(a, d) > \delta$. Thus, the nodes are too similar to the head node, and so the algorithm recursively repeats. In the second iteration, top down examines the nodes at the next level down. The left side contains no children, so the process ends and continues on the right side. Top down then considers the two lower children, and the algorithm continues.

considered if it has a unique starting point and is the shortest possible instance. Hawking calls these occurrences spans. More formally, a span is a unique, shortest string of words that contains all the query terms.

An example helps to elucidate the issue. The query terms for the following passage are *quick*, *brown*, and *fox*.

> The quick(1) brown(1)(2) fox(1)(2)(3) jumps over the lazy dog. The fast red bird flew over the jumping fox(4). The shining white moon rises over the flying bird. The quick(2)(3)(4) brown(3)(4) cow jumps over the rising moon.

The first span is labeled with (1), the second with (2), third with (3), and so on. Note how (1), (2), (3), and (4) all have unique starting points. In this case, then:

$$\begin{aligned}
|\sigma_{(q,1)}| &= 3 \\
|\sigma_{(q,2)}| &= 27 \\
|\sigma_{(q,3)}| &= 27 \\
|\sigma_{(q,4)}| &= 13
\end{aligned}$$

Hawking et al evaluated a number of proximity scoring formulas and determined that the best one uses a radical in the denominator.

$$P_{(q,d)} = \sum_{i=1}^{j} \frac{1}{\sqrt{|\sigma_{(q,i)}|}} \tag{5.6}$$

Using Equation 5.6, the proximity score for the passage above is 1.24.

Based on Hawking et al's recommendation, we chose to use Equation 5.6 for our implementation of proximity scoring. For more details on our implementation see Section 7.2. For our evaluation of proximity scoring, see Section 8.1 and Section 8.2.

Chapter 6

Testing

6.1 Query Testing

This chapter is an overview of the testing process. We give a detailed explanation of how we tested our three methods. We explain how we selected our set of queries, and also how we cached the pages for these queries. The process used in re-ranking is also explained. Finally, we describe how humans made pairwise comparisons of pages on each given query, and how we converted their results into an “ideal” human ranking of those pages.

6.1.1 Zipf’s Law

Zipf’s Law says when the frequencies of queries are ordered in decreasing order, then there is a certain relationship. This relationship is shown in a log-log plot, where the vertical axis shows the frequency of a given query, and the horizontal axis shows the queries ordered in decreasing order of frequency. That is to say, the query with the highest frequencies would be first, then the query with the second highest frequency, etc. If this plot follows Zipf’s Law, then it will have a slope of -1, as shown in our sample plot in Figure 6.1.

Generally, queries entered into search engines follow Zipf’s Law. When a set of queries follows Zipf’s Law, we can assume that it is representative of the web [zipfs].

6.1.2 Query List Generation

We used a total of fifty-eight queries in our precision testing, (Metric 1). Figure 6.2 lists our queries. These queries were methodically picked from two sets: one generated by the team, and another generated for us by Rosie Jones at Overture Services. Approximately ten queries were initially picked from the team’s set, and fifty more were selected from Overture’s set. The program we created to cache the pages did not account for instances where the query returned less than 200 pages. We discarded one query for this reason, and a second because of inappropriate content.

The reason why these queries were not randomly chosen from the two sets is that we wanted the set of queries to be representative of the web. Hence, we made sure that these

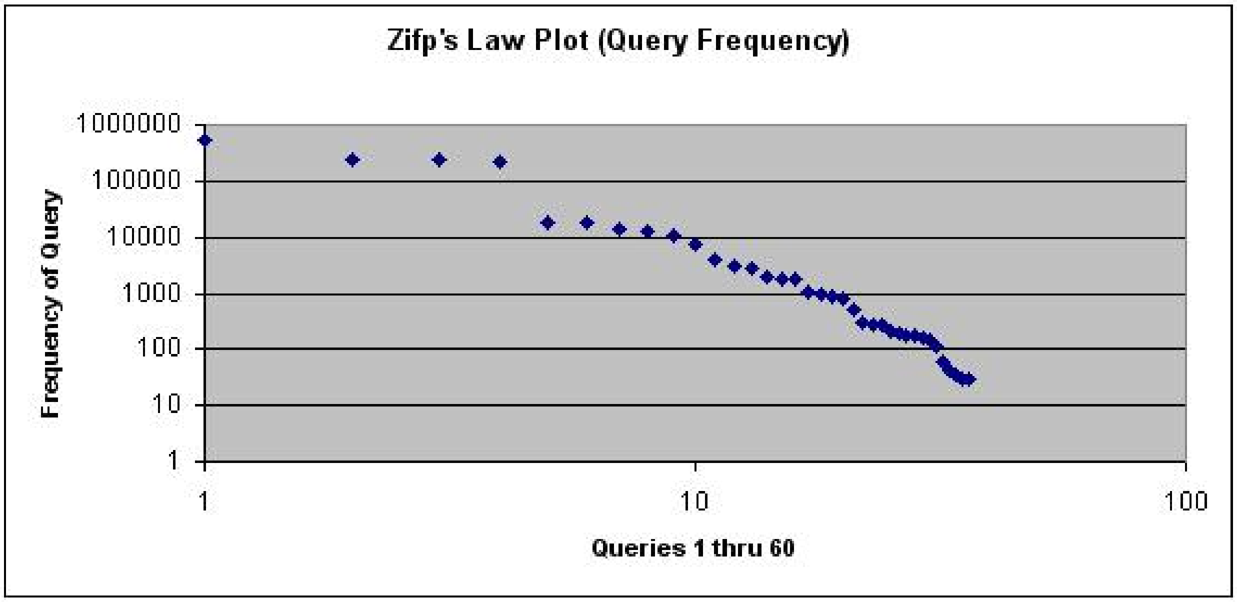


Figure 6.1: This is a log-log plot of frequency of query versus the descending order of frequencies of queries. This plot has a slope of −1, which means that the set of queries is said to follow Zipf's Law, and it is representative of the web.

queries followed Zipf's Law, which is explained above. See Figure 6.1 for a Zipf's Law plot of our initial set of sixty queries. The slope is approximately −1.

6.1.3 Caching and Re-ranking the Pages

The top 200 web pages for each of our fifty-eight queries were locally cached. These pages were run through each of our three methods. We took the union of the top ten returned results for each of the three methods and Google. Precision testing was done on this set. The number of web pages for each query ranged from sixteen to thirty-five web pages.

6.2 Human Testing

The team rated each of the fifty-eight queries for precision (Metric 1).

Also, in order to come up with the "ideal" human ranking for our relevance ordering metric (Metric 2), we recruited human testers to rank web pages for thirty of the fifty-eight queries. The testers were a total of twenty-eight Harvey Mudd College students, each of whom worked for an hour on one or two queries.

6.2.1 Precision Testing (Metric 1)

Precision is calculated for a given query by taking the fraction of web pages returned from a search engine that are deemed relevant by humans (see Section 4.2.1). For each of the fifty-eight queries, the team went through the web pages, and scored each page as either *relevant*

a summary on world war one	**learn thai massage in thailand**
always on top utility	**los molinos**
bartender drinks mixed drink recipes	lyrics eminem super man
blue mountain	manual transmission near phase
boston apartments	**maternity scrubs**
brand new rock	mindstorm lego robotics
certification in housing	**motorola v120c phone accessories**
cheap tech gear	**mystery party**
church service	nt administration helpdesk
colleges that offer training for private	**paint stick new inventions**
contour pillow	pokemon
cybernet ventures	**pretty sunset landscapes**
eddie bauer infant car seat	san luis obispo
elva corporation	sebring headlight
fleet space museum calif	**sesame street cookie monster**
forklift operator	she loves me and veronica
free chess tactics	**sick building syndrome**
front bumper	star trekker
gender differences in education	**statistical language analysis**
guitar hanger stand	structural design context applications
harry potter	sun city arizona tours to laughlin and vegas
health insurance companies of tampa bay florida	super child model
health insurance probability and accountability act	**terrorism iraq afghanistan**
infectious diseases	the actors gang
java	**the psychological corporation**
john milton	**thomas the tank boco**
jpeg cd burner slide show dvd player	**u.s.a olympic ice skating contenders**
kingdom hearts 59 puppies	**un members**
laptop cases	world junior hockey

Figure 6.2: Above are fifty-eight of the initial sixty queries that we used in testing for precision (two of the queries were filtered out). The words in bold are the queries used in testing for relevance ordering.

or *not relevant* to the query. Two different people (on the team) rated the web pages for each query. If a web page was given a relevant score by one or both of these people, then the web page was deemed relevant for that query. Otherwise the web page was considered irrelevant for that query. A screenshot of the program we used to rate web pages is given below in Figure 6.3.

6.2.2 Testing for Relevance Ordering

In order to determine the relevance ordering for a given query, human testers made pairwise comparisons of the web pages returned by our three methods and Google. Then for each of the queries, these comparisons were plugged into a matrix. The Rayleigh Method was then used to find a ranking vector. Kendall's Tau was then applied to each of the ranking vectors from the three methods and Google. Kendall's Tau returned a score between -1 and 1, where the higher the number, the better the relevance ordering. For each query, up to $\binom{n}{2}$ comparisons are made, where n is the total number of web pages for that query (see Section 4.2.1 and Section 4.2.2 for more details on our metrics).

6.2.2.1 Human Testing and Interface

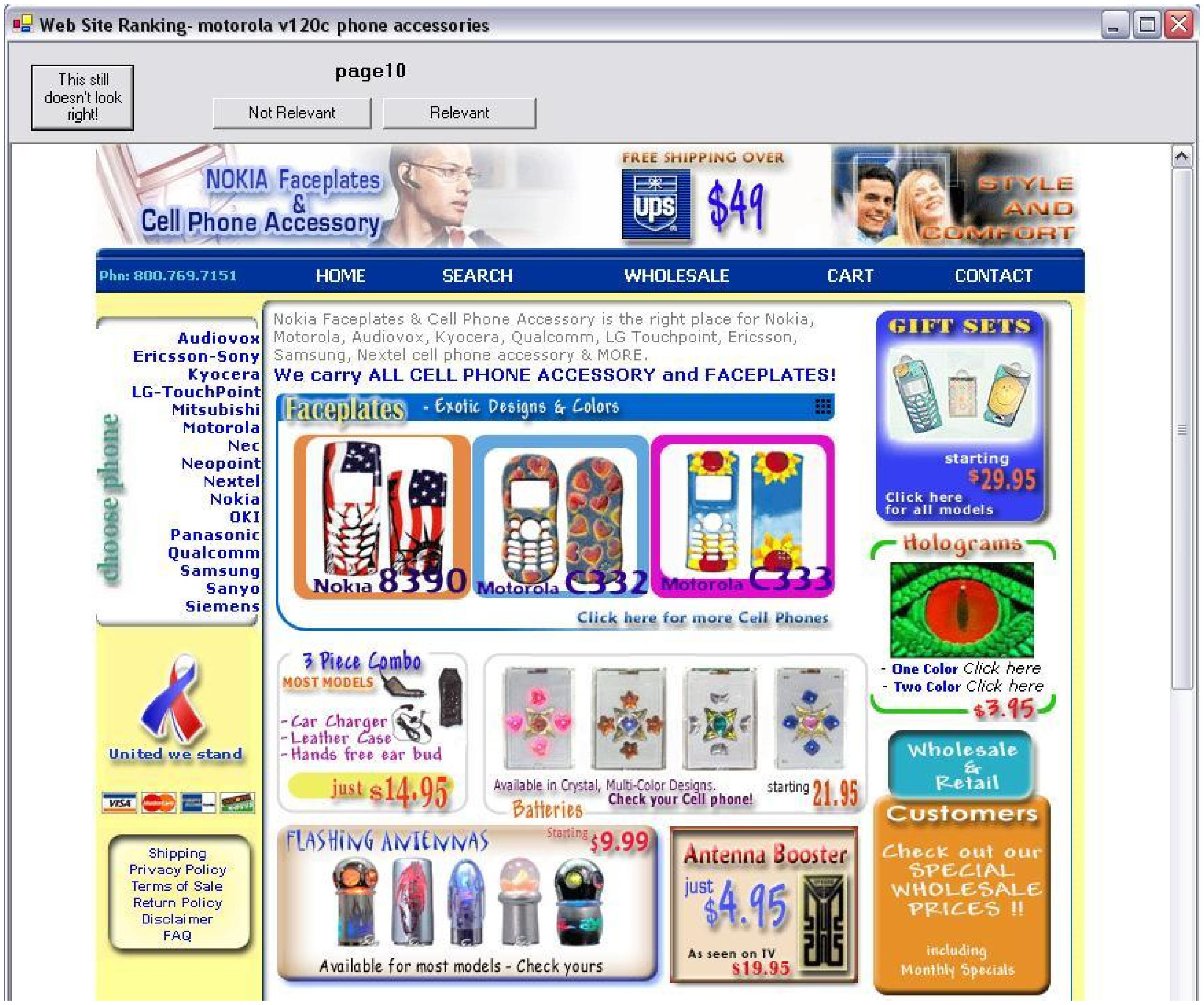
Thirty out of the fifty-eight queries were randomly selected. This smaller set was checked to ensure that it still followed Zipf's Law(See Figure 6.1). Again, for each query, we took the union of the top ten returned results for the three methods and Google. Then, we had this set tested by humans for relevance ordering in order to create the "ideal" ranking vector. This was done by taking all the returned pages for each query, and having them compared against each other. A screenshot of the testing interface is shown in Figure 6.4. The results were then plugged into a matrix, on which we ran the Rayleigh method to find the eigenvector of the largest eigenvalues. The Perron-Frobenius Theorem was used here (Below is a description of the Perron Frobenius Theorem). After this was done, we used Kendall's Tau to compare each of the three methods and Google to the "ideal" ranking vector (See Section 6.2.2.4).

6.2.2.2 Discussion of the Perron-Frobenius Theorem

The Perron-Frobenius theorem told us when there would be a unique solution when trying to rank the web pages using the pairwise comparisons.

Theorem *If the (nontrivial) matrix A has nonnegative entries, then there exists an eigenvector r with nonnegative entries, corresponding to a positive eigenvalue λ. Furthermore, if the matrix A is irreducible, the eigenvector r has strictly positive entries, is unique and simple, and the corresponding eigenvalue is the largest eigenvalue of A in absolute value [Kee].*

To understand when A is irreducible, we first define the idea of rows *communicating* with


Web Site Ranking- motorola v120c phone accessories

Nokia Faceplates & Cell Phone Accessory is the right place for Nokia, Motorola, Audiovox, Kyocera, Qualcomm, LG Touchpoint, Ericsson, Samsung, Nextel cell phone accessory & MORE.

We carry ALL CELL PHONE ACCESSORY and FACEPLATES!

Figure 6.3: This is a screenshot of the interface we used in order to score whether or not a web page is relevant to a given query. In the top center of the interface we have the buttons "relevant" and "not relevant". When either of these are clicked, then the program goes on to the next web page. This web page was considered to be relevant by both raters for the query, "motorola v120 phone accessories," which is indicated in the top left corner. For a full explanation of the user interface, see Section 7.5.

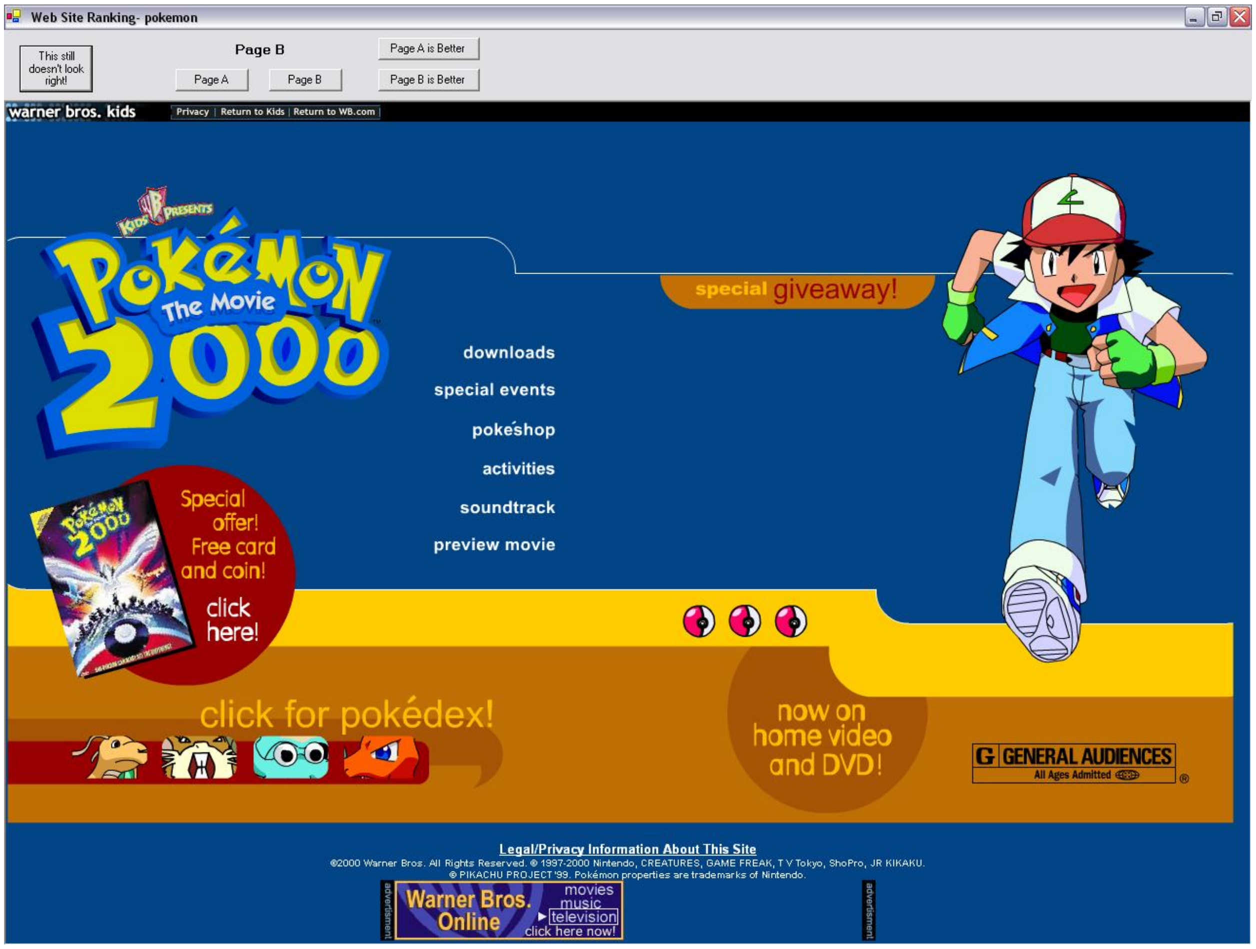

Web Site Ranking- pokemon
This still doesn't look right!
Page B
Page A
Page B
Page A is Better
Page B is Better
warner bros. kids
Privacy | Return to Kids | Return to WB.com
Pokémon The Movie 2000
special giveaway!
downloads
special events
pokéshop
activities
soundtrack
preview movie
Special offer! Free card and coin! click here!
click for pokédex!
now on home video and DVD!
G GENERAL AUDIENCES All Ages Admitted
Legal/Privacy Information About This Site
Warner Bros. Online
movies music television click here now!

Figure 6.4: The above picture is a screenshot of the interface that we built in order to allow testers to make pairwise comparisons. The query in this case is, "pokemon," which is written on the top left corner of the snapshot. The user is able to flip back and forth between this page (Page B), and another page (Page A). If Page A is more relevant to the query then the user clicks on the button "Page A is Better". Otherwise, "Page B is Better" is selected. When either of these buttons are clicked, then the program goes on to the next comparison, and continues on until most of the web pages for the query, "pokemon," are compared. For more details on the user interface, go to Section 7.5.

other rows.

Definition *A Row i of matrix A communicates with row j if there exists a set of rows $R = r_1, r_2, r_3, \dots r_n$ such that $A[i, r_1] > 0, A[r_1, r_2] > 0, A[r_2, r_3] > 0, \dots A[r_n, j] > 0$.*

For example, if $A[3, 4] = 1$ and $A[4, 9] = .5$ then we can say that row 3 *communicates* with row 9. We say that A is *irreducible* if every row communicates with every other row.

We created an $n \times n$ matrix A, where $A[i, j] = 1$ if web page i was deemed more relevant than web page j by human testings, and 0 otherwise. The Perron-Frobenius theorem tells us that the web pages, which correspond to the rows, will be ranked and have a unique solution corresponding to the largest eigenvalue, λ of $|A|$ if A is irreducible.

6.2.2.3 Applying the Rayleigh Method

Keener explains that we can use the Rayleigh Method to find the eigenvector of the largest eigenvalue of A, assuming the matrix A is irreducible. The Rayleigh Method states that

$$\lim_{n \to \infty} \frac{A^n r_0}{|A^n r_0|} = r, \tag{6.1}$$

for any nonnegative vector r_0, where $r^{\rightharpoonup}$ is the eigenvector corresponding to the largest eigenvalue. Applying the Rayleigh method without checking for irreducibility, we found that the eigenvector returned for some queries had rows with scores of 0. We determined that this was because the matrix was not irreducible; there were certain web pages that were categorically more relevant than others, and so the communication criterion for irreducibility did not hold. Knowing this, we partitioned A into matrices that were irreducible amongst their own respective rows. This meant separately evaluating those rows who initially received scores of 0 in the final eigenvector. Then, the rankings of these rows were appended to the bottom of the rows from the original non-zero rankings. This method ran recursively to ensure that all rows that received a zero score were ranked effectively. This final vector represented the ideal human ranking that we so eagerly sought.

6.2.2.4 Applying Kendall's Tau

We took the ranking from our modified version of the Rayleigh method and used it to find Kendall's Tau (See Section 4.2.3). For each query, each of the results from our three methods and the algorithmic search engine results were re-indexed according to the order of our ideal human ranking. Using Kendall's Tau, we then computed the correlation between the ideal human ranking and each of the four rankings.

The *ith*entry in the eigenvector corresponding to λ is the score of the *ith* page. Ordering the web pages in descending order of their scores gives the "idea" human ranking.

Chapter 7

Programs and Source Code

Throughout the course of this project we wrote many programs to assist our project. While the source code is self-documenting, this section presents a high-level overview of the code – what it does, and what it does not do. Also, we identify the known bugs and issues, such as ineffeciency.

The most significant programs are `miuanalysis.exe` and `proxscore.pl` to perform MIU analysis and proximity scoring, respectively. These programs are implementations of the theory discussed in Section 5.1 and Section 5.2, respectively. For these programs we present excerpts from the source code in addition to describing the general behavior and identifying any issues or bugs. The source code for `miuanalysis.exe` begins on page 39 and ends on page 51. The source code for `proxscore.pl` begins on page 52 and ends on page 55.

For the other programs we only sketch their behavior and direct the interested readers to the source code itself which is very comprehensive. The CD accompanying this report contains the full source code.

7.1 MIU Analysis

Our MIU analysis program, `miuanalysis.exe`, performs MIU analysis on a web page using either bottom up or top down MIU analysis. Concisely, `miuanalysis.exe` is a command line program which takes an HTML file as input, and outputs the page with the MIUs delineated by comments in the HTML. The program is written in ANSI C and uses the TidyLib HTML parsing library [TidyLib] and the libmba data structure library [libmba]. Although we have only compiled our program using Visual Studio .NET to the Win32 platform, the program does not use any features exclusive to this platform. Consequently, porting the program to the Linux platform or other platforms is both feasible and easy.

Using `miuanalysis.exe` is simple. The program has a few command line options, such as the type of MIU analysis and the filename. It outputs the MIU analysis on `STDOUT`. It also prints any extraneous information on `STDERR`.

```
usage: miuanalysis.exe type htmlfile [options]

type       -td or --topdown for top down processing
```

```
           -bu or --bottomup for bottom up processing
htmlfile   the file on which to perform MIU analysis

options
--config file or -cfg file where file is the path to the configuration file
```

The next sections describe the types of files used by `miuanalysis.exe`, known issues, overall structure, and details of the bottom up and top down algorithms.

7.1.1 Files

There are four types of files used by `miuanalysis.exe`: configuration files, stoplist files, HTML files, and MIU files.

7.1.1.1 Configuration Files

Configuration files indicate extra options for MIU analysis. They are text files that use key-value statements of the form: `key = value`. Figure 7.1 shows the configuration file used in our experiment for bottom up MIU analysis.

```
bu_root = html
minlength = 20
```

Figure 7.1: The configuration file `bottomup.cfg` used for bottom up MIU analysis. This file instructs bottom up analysis to use the <html> tag as the root of the DOM tag tree and specifies the minimum MIU length is 20 words. The minimum length requires that MIUs are at least x words, if possible.

In total, there are 11 possible keys, listed below.

`verbose`	*integer*	Controls the verbosity of the program. The possible values are 0 - 4. 0 prints no extra information, 4 prints all possible extra information about the MIU analysis process. All extra output is printed on `STDOUT`. The default value is 1.
`stemming`	*string*	Controls the use of stemming in the program. A value of `yes` always uses stemming; a value of `no` never uses stemming. The default value is `yes`.
`stoplist`	*string*	Controls the file used as a stoplist. If this value is not `none` then this option specifies the path to the stoplist file to use. If the value is `none` then no stoplist is used. For details on the stoplist file format, see the next section. The default value is `stoplist.txt`.

`minlength`	*integer*	This option specifies the minimum length of MIUs and is currently only used by the bottom up algorithm. While any positive integer value is legal, MIU analysis will only produce valid results if this value is reasonable, i.e . The default value is 5.
`bu_delta`	*integer*	In Li et al's algorithm, δ is the number of common display elements two nodes must have for the bottom up algorithm to merge them. We maintain Li et al's notation, and `bu_delta` controls this value for our implementation. Any positive integer value is legal, although unreasonable values generate unreasonable results. The default value, which Li et al suggest, is 3.
`bu_omega`	*integer*	In Li et al's algorithm, ω is the number of common words two nodes must have for the bottom up algorithm to merge them as adjacent paragraphs. We maintain Li et al's notation, and `bu_omega` controls this value for our implementation. Any positive integer value is legal, although unreasonable values generate unreasonable results. The default value, which Li et al suggest, is 2.
`bu_lengthlast`	*string*	With our modifications to Li et al's algorithm, we added a condition that MIUs must be of a minimum length. Any value for this key signals the algorithm to apply the length criterion after completing Li et al's merging criteria. If the key is not present, the algorithm applies the length criteria first, before Li et al's merging criteria.
`bu_root`	*string*	There are three places the bottom up algorithm can start MIU analysis. This option controls the location the algorithm chooses. If the value is `root` then the algorithm starts at the root of the DOM tree; if the value is `html` then the algorithm starts at the <html> tag; and if the value is `body` then the algorithm starts at the <body> tag. The default value is `root`.
`td_delta`	*float*	In the top down algorithm, the δ parameter determines when the algorithm splits a node because it is too similar to the overall document. The option `td_delta` corresponds to this top down parameter. Possible values are floating point numbers between 0 and 1. The default value is 0.40.

`td_root`	*string*	There are three places the top down algorithm can start MIU analysis. This option controls the location the algorithm chooses. If the value is `root` then the algorithm starts at the root of the DOM tree; if the value is `html` then the algorithm starts at the <html> tag; and if the value is `body` then the algorithm starts at the <body> tag. The default value is `root`.

7.1.1.2 Stoplist Files

Stoplist files are text files that list words for a stoplist. Text processing algorithms frequently use stoplists to eliminate common words such as "the", "a" and "and." In our file, newline characters seperate words. Lines that begin with the characters `!` or `#`are comments. Thus, the following file specifies a stoplist with five words, *the*, *a*, *an*, *that*, *then*.

```
# Sample stoplist file.
the
a
an
# but -- a commented word, and not included in the list
that
then
! there -- a commented word, and not included in the list
# End sample stoplist file.
```

The program uses the stoplist file specified by the `stoplist` option in the configuration file.

7.1.1.3 HTML Files

`miuanalysis.exe` can take in as input any HTML file, with a few exceptions which are noted in Section 7.1.2.

7.1.1.4 MIU files

MIU files are the output of the `miuanalysis.exe` program. These files contain information about the MIUs and their location within an HTML document.

7.1.2 Unresolved Issues

Unfortunately, `miuanalysis.exe` contains a number of flaws. While none of these inhibited the program or the project, we identify and discuss them in this section, so that any future developers will know what does not work or needs improvement.

The first flaw discussed is a problem with the TidyLib library and HTML file (Section 7.1.2.1). The second flaw is a restriction on synonymous attributes in HTML file (Section 7.1.2.2). The final flaw is an inefficiency in the `MIU_GetText` function (Section 7.1.2.3).

7.1.2.1 HTML Parsing with Tidy

In virtually every case, the TidyLib library parsed the HTML file into an internal tag tree structure. However, we identified two cases that caused the library to fail during parsing. We hope the developers of the library will fix these bugs in future versions.

The first failure case occurred on `page9.html` from the query "health insurance probability and accountability act." On this page, the first <html> tag declaration contained an error. The line from the file that caused this error follows.

```
<HTML xmlns>
```

In this case, TidyLib attempted to get the value of the `xmlns` attribute. The internal call returns a `null` value which the library does not catch, and the program fails. To address this issue, we removed the `xmlns` attribute from the <html> tag in `page9.html`.

The second failure occurred on `page52.html` and `page185.html` for the query "health insurance probability and accountability act." These files contained an error for the `style` attribute in various places. In particular, they had multiple HTML tags with two style attributes, like the following excerpt from `page52.html`.

```
<SPAN STYLE="font-size: 11pt" STYLE="">
```

When TidyLib parses this input, it tries to concatenate the style attributes into a single attribute. However, the library did not check for an empty second attribute and fails when this occurs. In this case, we removed the offending style attribute from the two files.

7.1.2.2 Synonymous Attributes

One of the functions for Li et al's bottom up MIU analysis algorithm required comparing attributes between two nodes in the tag tree. In our implementation of this function, we do not resolve synonymous attributes, that is, equivalent attributes that have multiple forms. For example, the color attribute of a <font> tag can be either the name of a color or a hexadecimal value and the following tags have the same attributes.

```
<font fact="arial" color="0x008000">
<font face="arial" color="green">
```

Currently, our tag attribute comparison algorithm does not convert synonymous tags to a common form. Hence, although Li et al's algorithm suggests that those tags should have two common attributes, our function only performs a textual comparison of the attribute values and determines the tags have only one common attribute. This limitation is noted in the comment preceding the `MIU_BU_DisplaySimilarity` function (in `bu_eval.c`).

The most recent versions of TidyLib contain details on mapping between textual colors and their hexadecimal equivalents. Any improvements could use this functionality to resolve synonymous attributes, at least in the case of colors.

7.1.2.3 Inefficient Memory Usage in `MIU_GetText`

While we made every attempt to keep `miuanalysis.exe` as efficient as possible, one of the major deficiencies is the `MIU_GetText` function (from `mius.c`). This function retrieves the text for an MIU node and caches the result. In order to quickly code this function, we used a simple recursive descent and concatenation algorithm to build the text. Consequently, the `MIU_GetText` function dynamically allocates and frees portions of memory, which are potentially large

While this works well for bottom up MIU analysis – after the first call, all of the subsequent calls are cached – it is inappropriate for our top down MIU analysis algorithm. We were unable to cache the results for undifferentiated MIUs, that is, what the top down algorithm tries to split. Each node, then, must be visited many times to get the appropriate text in top down MIU analysis.

A better implementation of the function would retrieve the text of the entire document into a single buffer. Each node would then store pointers to its portion of the global buffer. TidyLib uses this method internally. In particular, this method would greatly simplify concatenation of adjacent nodes, which frequently occurs in bottom up MIU analysis. Instead of allocating a new block of memory, the concatenated node simply stores the start pointer from the first node and the end pointer from the second node.

7.1.3 Overall Structure

In this section, we outline the overall structure of the source code for `miuanalysis.exe`. The intention is to explain the internal structure of the program to make reading and comprehending the code easier.

The source code for `miuanalysis.exe` is divided into discrete units. These units are largely independent building blocks. The main programs and algorithm combine these building blocks into the full program. There are four key units in the code.

- `text` - This unit includes functions to compute textual similarity, stem words, and manage a stoplist.
- `miu` - The main MIU unit which contains functions to create, delete, merge, and analyze MIUs.
- `miu_bu` - The bottom up MIU analysis unit. This unit heavily uses the `miu` unit and only contains one function to apply the bottom up algorithm to a web page.
- `miu_td` - The top down MIU analysis unit. This unit is analogous to the `miu_bu` unit for the top down algorithm.

Units in the code can contain private functions that are not globally available. In many cases, we accomplish this using the `static` modifier, such as for the `text` unit. In other cases, there are private internal header files, such as for the `miu_bu` and `miu_td` units.

Functions that belong in a unit contain the unit name in their function name. For example, the function `MIU_GetText` retrieves the text for an MIU and is in the `miu` unit. Likewise, the function `TEXT_Stem` stems a word and is in the `text` unit.

The unit name of a function determines which file it is defined in. For example, functions in the `text` unit are defined in `text.c` and prototyped in `text.h`.

The next two sections show how the bottom up and top down algorithms use the unit structure to merge and split MIUs.

7.1.4 Bottom Up

The bottom up MIU analysis algorithm is explained in Section 5.1.2. In this section, we discuss the critical code path for bottom up MIU analysis. This path involves the functions `MIU_BU_BuildMIUs` and `ConstructMIUs_LengthFirst` in `bu_build.c`. The documentation on these functions in largely inline with the code and reproduced below. `MIU_BU_BuildMIUs` calls the function `BuildMIUsAtDepth` which recurses down the tag tree until it reaches the desired depth. It then calls `ConstructMIUs_LengthFirst` on each node at that depth.

Our major deviation from Li et al's algorithm is our minimum length criteria. We first merge adjacent nodes together until the merged nodes are greater than the value in the `min_length` variable. This corresponds with step 0 in the `ConstructMIUs_LengthFirst` function. Li et al's merging equations, Equations 5.1, 5.2, and 5.3, correspond to the functions `HeaderAndContent`, `HeaderAndContentPair`, and `AdjacentParagraphs`, respectively.

There is one non obvious aspect about the code. Although the `ConstructMIUs` function contains syntax which implies an infinite loop, this does not occur because the final statement of the loop iteration causes the loop to end.

```
for(itr = 0; itr >= 0; itr++)
{
     ...
     break;
}
```

The two functions `MIU_BU_BuildMIUs` and `ConstructMIUs_LengthFirst` are reproduced below.

The code listing ends on page 46.

```
/*
 * MIU_BU_BuildMIUs applies the algorithm from Li et al+ to merge MIUs.
 *
 * + see Xiaoli Li, Tong-Heng Phang, Minqing Hu, and Bing Liu. Using micro
 *       information units for Internet search. In Proceedings of the Eleventh
 *       International Conference on Information and Knowledge Management,
 *       pages 566573. ACM Press, 2002.
 *
 * input:
 *  tdoc - the tidy document for the webpage
 *  root - the root of the miu tree
 *
 * output:
 *  root - the revised miu tree where the mius are the leaf nodes
 *
 * returns:
```

```
 *  TRUE for success, FALSE for failure
 */

BOOL MIU_BU_BuildMIUs(IN TidyDoc tdoc, IN OUT miu_node_t *root)
{
    int tree_depth = 0;
    int depth = 0;
    BOOL rval = TRUE;
    int value = 0;
    bu_stats_t stats = {0};
    char val_buffer[STRING_SHORT];

    // parameter validation
    assert(tdoc);
    assert(root);

    if (!tdoc || !root)
    {
        debug_printf(DEBUG_EXTENSIVE, "invalid parameters to MIU_BU_BuildMIUs\n");
        return (FALSE);
    }

    // retrieve values of delta and omega from the config file
    [excluded from the listing...]

    //
    // since this is a bottom-up algorithm, the analysis starts by examining
    // all of the lowest level of the tree
    //

    tree_depth = MIU_TreeDepth(root);
    debug_printf(DEBUG_EXTENSIVE, "tree depth %i\n", tree_depth);

    for (depth = tree_depth - 1; depth > 0; depth--)
    {
        debug_printf(DEBUG_EXTENSIVE, "building MIUs for level %i\n", depth);
        if (!BuildMIUsAtDepth(tdoc, root, depth, &stats))
        {
            error_printf(ERR_MIU_BUILD_FAILED_DEPTH, depth);
            rval = FALSE;
            break;
        }
    }

    debug_printf(DEBUG_BASIC, "number of length mergings: %i\n"
                              "number of header mergings: %i\n"
                              "number of paragraph mergings: %i\n",
                   stats.num_length, stats.num_header, stats.num_paragraph);

    return (rval);
}

/*
```

```
 * ConstructMIUs_LengthFirst applies the merging algorithm which does
 * four things:
 *
 * 0.  Merge all nodes of insufficient length.
 * 1.  Merge header and content paragraphs (as adjacent nodes in the tree).
 * 2.  Merge adjacent text paragraphs.
 * 3.  Merge singleton children with their parents.
 *
 * THIS CAN ONLY RUN IF ALL THE NODES ARE LEAF NODES.
 *
 * This is the length first merging algorithm.  See ConstructMIUs_LengthLast
 * for a version that moves the length merging to the end of the algorithm.
 *
 * input:
 *  tdoc - the tidy document, this is needed to get the node text
 *  node - a single node at the merge level
 *  stats - a statistics structure to count the applications of the various
 *          merging criteria
 *
 * output:
 *  node - node is not guaranteed to be valid after the call
 *  stats - the updated statistics structure with all the mergings from the
 *          current call and all recursive calls
 *
 * returns:
 *  TRUE for success, FALSE for failure
 */

static BOOL ConstructMIUs_LengthFirst(IN TidyDoc tdoc, IN OUT miu_node_t *node,
                                      IN OUT bu_stats_t* stats)
{
    BOOL rval = FALSE;
    int itr = 0;

    // parameter validation
    assert(tdoc);
    assert(node);
    assert(stats);

    //
    // 0.  Check MIU length for automerge, but we need at least 1 sibling
    //
    if (MIU_SiblingCount(node) > 0)
    {
        miu_node_t *mn = node;
        int length = 0;

        while (mn)
        {
            // check if the MIU is too short
            if (linkedlist_is_empty(mn->child_miunodes) &&
                MIU_BU_TextLength(tdoc, mn, &length) &&
                length <= min_length)
            {
                if (mn->next && linkedlist_is_empty(mn->next->child_miunodes))
                {
                    // handle on special case
                    if (mn == node)
                    {
                        if (!MIU_MergeAdjacent(mn, mn->next, &node))
                        {
                            return (FALSE);
                        }

                        // set mn to the new merged node.
                        mn = node;
                    }
                    else
                    {
                        if (!MIU_MergeAdjacent(mn, mn->next, &mn))
                        {
                            return (FALSE);
                        }
                    }
                }
                else if (mn->prev && linkedlist_is_empty(mn->prev->child_miunodes))
                {
                    // again, we have to handle to case where
                    // mn->prev == node

                    if (mn->prev == node)
                    {
                        if (!MIU_MergeAdjacent(mn->prev, mn, &node))
                        {
                            return (FALSE);
                        }

                        mn = node;
                    }
                    else
                    {
                        if (!MIU_MergeAdjacent(mn->prev, mn, &mn))
                        {
                            return (FALSE);
                        }
                    }
                }
                else
                {
                    // in this case, it's a singleton node
                    // because of merging, so break out
                    break;
                }

                // we need to make sure this node is long enough too
                // so we don't want to step the node
```

```
                //
                // note that this continue refers to the
                // the while(nm) loop over the automerge length nodes
                stats->num_length++;
                continue;
            }

            // step the node
            mn = mn->next;
        }
    }

    if (!VerifyLeafNodes(node))
    {
        // this is simple, because the algorithm doesn't apply when
        // all the nodes are not leaf nodes
        return (TRUE);
    }

    //
    // count the number of iterations
    // all continue statement should refer to THIS loop
    //

    for (itr = 0; itr >= 0; itr++)
    {
        //
        // 1.  Merge header and content paragraphs.
        //
        if (node->next)
        {
            miu_node_t *mna = node;
            miu_node_t *mnb = node->next;
            // we need to make sure the "next" node was valid.

            if (MIU_SiblingCount(mnb) == 1 &&
                HeaderAndContent(tdoc, mna, mnb))
            {
                // if there are only two nodes, which are a header and content,
                // then merge them.
                if (!MIU_MergeAdjacent(mna, mnb, &node))
                {
                    return (FALSE);
                }

                stats->num_header++;
            }
            else if (HeaderAndContent(tdoc, mna, mnb))
            {
                // if there are any nodes c/d such that
                // displaySimilarity(A, C) >= delta &&
                // displaySimilarity(B, D) >= delta, then
                // merge A, B, and all such C, D.
                miu_node_t *mnc = mnb->next;
                miu_node_t *mnd = NULL;
                BOOL merge_ab = FALSE;

                while (mnc)
                {
                    mnd = mnc->next;
                    if (mnd)
                    {
                        if (HeaderAndContentPair(tdoc, mna, mnb, mnc, mnd))
                        {
                            merge_ab = TRUE;

                            debug_printf(DEBUG_VERBOSE, "merging MIU {%s} ",
                                 MIU_PrintNode(tdoc, mnc));
                            debug_printf(DEBUG_VERBOSE, "{%s} hc\n",
                                 MIU_PrintNode(tdoc, mnd));

                            if (!MIU_MergeAdjacent(mnc, mnd, &mnc))
                            {
                                return (FALSE);
                            }

                            stats->num_header++;
                        }
                        else
                        {
                            // move onto the next pair
                            mnc = mnd;
                        }
                    }
                    else
                    {
                        // change mnc so we terminate
                        mnc = NULL;
                    }
                }

                if (merge_ab)
                {
                    debug_printf(DEBUG_VERBOSE, "merging MIU {%s} ",
                                 MIU_PrintNode(tdoc, node));
                    debug_printf(DEBUG_VERBOSE, "{%s} hc\n",
                                 MIU_PrintNode(tdoc, mnb));

                    if (!MIU_MergeAdjacent(node, mnb, &node))
                    {
                        return (FALSE);
                    }

                    stats->num_header++;
                }
            }
        }

        //
        // 2.  Merge adjacent text paragraphs
        //
        if (node->next)
        {
            miu_node_t *mnx = node;
            miu_node_t *mny = node->next;
            BOOL merged = FALSE;

            while (mny)
            {
                if (AdjacentParagraphs(tdoc, mnx, mny))
                {
                    debug_printf(DEBUG_VERBOSE, "merging MIU {%s} ",
                                    MIU_PrintNode(tdoc, mnx));
                    debug_printf(DEBUG_VERBOSE, "{%s} hc\n",
                                    MIU_PrintNode(tdoc, mny));

                    if (!MIU_MergeAdjacent(mnx, mny, &mny))
                    {
                        return (FALSE);
                    }

                    stats->num_paragraph++;

                    merged = TRUE;

                    break;
                }

                // increment the nodes
                mnx = mny;
                mny = mny->next;
            }

            // if there was a merging, restart the process and
            // let's try it again!
            if (merged)
            {
                continue;
            }
        }

        //
        // 3.  Merge singleton children with their parents.
        //

        // a sibling count of 0 indicates NO siblings, i.e. a singleton node
        if (MIU_SiblingCount(node) == 0)
        {
            debug_printf(DEBUG_VERBOSE, "merging MIU {%s} up\n",
                MIU_PrintNode(tdoc, node));
            MIU_MergeUp(node);
        }


        rval = TRUE;
        break;
    }

    return (TRUE);
}
```

7.1.5 Top Down

The top down MIU analysis algorithm is explained in Section 5.1.3. In this section, we discuss the critical code path for top down MIU analysis. This path involves the functions `MIU_TD_BuildMIUs` and `SplitMIUs`.

The function `SplitMIUs` recursively divides the tag tree into MIUs. While the documentation indicates that the code implements the dual threshold top down MIU analysis algorithm, see Section 10.2.1, the actual implementation is of the original top down algorithm.

One interesting notes about the code follow. The algorithm uses the `TEXT_Similarity` function to compute the similarity metric described by Equation 5.4. That function takes two hashtables which contain a word to frequency map. The result is a floating point value from 0.0 to 1.0 describing the similarity between the two text passages.

We reproduce the two functions `MIU_TD_BuildMIUs` and `SplitMIUs` below.

The code listing ends on page 51.

```
/*
 * MIU_TD_BuildMIUs constructs MIUs for the tag tree using the top down
 * algorithm.
 *
 * input:
 *  tdoc - the tidy document
 *  root - the root miu or document node
 *
 * output:
 *  root - the root document node with MIUs created at child levels
 *
 * returns:
 *  TRUE for success, FALSE for failure
 */

BOOL MIU_TD_BuildMIUs(IN TidyDoc tdoc, IN OUT miu_node_t *root)
{
    // variables
    const char *doctext = NULL;
    BOOL rval = FALSE;
    // check parameters
    assert(tdoc);
    assert(root);

    if (!tdoc || !root)
    {
        return (FALSE);
    }

    // load values for delta and epsilon
    [excluded from listing...]

    // get the document text
    if (MIU_GetText(tdoc, root, &doctext, NULL))
    {
        struct hashmap *docfreq = NULL;
        int totalwords = 0;

        if (TEXT_WordFrequencies(doctext, &docfreq, &totalwords))
        {
            td_stats_t stats = {0};

            rval = SplitMIUs(tdoc, root, docfreq, &stats);

            hashmap_del(docfreq);
        }
    }

    return (rval);
}

/*
 * SplitMIUs performs the top down splitting algorithm.  The algorithm
 * on node A works as follows:
 *
 * For each child a of node A, construct an MIU node for a and
 * compute the text similarity between a and the document.
 *
 *  (1) if sim(a) > delta, call SplitMIUs(a) and continue.
 *  (2) if sim(a) < delta, sim(a) > epsilon, then continue.
 *  (3) if sim(a) < epsilon, then check if the similarity between
 *        a and either of its neighbors is greater than delta,
 *        if so, merge a and its neighbor.
 *
 * input:
 *  tdoc - the tidy document
 *  node - the MIU node
 *  docfreq - the frequency of words in the document
 *  stats - the splitting statistics
 *
 * output:
 *  node - the MIU node with child MIUs constructed
 *  stats - the updated splitting statistics
 *
 * returns:
 *  TRUE for success, FALSE for failure
 */
static BOOL SplitMIUs(IN TidyDoc tdoc, IN OUT miu_node_t *node,
                      IN struct hashmap *docfreq, IN OUT td_stats_t *stats)
{
    BOOL rval = TRUE;
    TidyNode tnode = NULL;
    TidyNode tchild = NULL;

    assert(tdoc);
    assert(node);
    assert(docfreq);
    assert(stats);

    //
    // first, segment the children into MIUs
    //

    tnode = linkedlist_get(node->tnodes, 0);
    if (!tnode)
    {
        error_printf(ERR_NO_NODE_IN_LINKEDLIST);
        return (FALSE);
    }

    for (tchild = tidyGetChild(tnode); tchild; tchild = tidyGetNext(tchild))
    {
        miu_node_t *mnode = MIU_NewMIUNode();
        if (mnode)
        {
            // set the parent
            mnode->parent = node;

            // add the child tnode to the decendent
            // add the decendent to the list of children
            if (linkedlist_add(mnode->tnodes, tchild) &&
                linkedlist_add(node->child_miunodes, mnode))
            {
                rval = TRUE;
            }
            else
            {
                error_printf(ERR_APPEND_NODE_FAILED);
                rval = FALSE;
                break;
            }
        }
        else
        {
            error_printf(ERR_NODE_ALLOC_FAILED);
            rval = FALSE;
```

```
                break;
            }
        }

        // now we need to assign next/prev pointers in the MIUs
        if (rval && !linkedlist_is_empty(node->child_miunodes))
        {
            // assign next/prev pointers to children, but only if
            // there is at least one child node
            miu_node_t* prev = NULL;
            miu_node_t* cur = NULL;
            miu_node_t* next = NULL;

            // assign prev pointers
            linkedlist_iterate(node->child_miunodes);
            prev = linkedlist_next(node->child_miunodes);

            for (cur = linkedlist_next(node->child_miunodes);
                 cur; prev = cur, cur = linkedlist_next(node->child_miunodes))
            {
                cur->prev = prev;
            }

            // now cur is null, but prev was the last valid element
            next = prev;

            // look backwards through the chain to assign next pointers
            while (cur = next->prev)
            {
                cur->next = next;
                next = cur;
            }
        }

        //
        // now we iterate over the child_miunodes
        //

        if (rval)
        {
            miu_node_t *child = NULL;
            linkedlist_iterate(node->child_miunodes);
            while (child = linkedlist_next(node->child_miunodes))
            {
                //
                // compute the similarity between the child
                // and the document
                //

                const char *text = NULL;
                struct hashmap *childfreq = NULL;

                if (MIU_GetText(tdoc, child, &text, NULL) && text &&
                    TEXT_WordFrequencies(text, &childfreq, NULL))
                {
                    float sim = TEXT_Similarity(docfreq, childfreq);

                    //
                    // apply condition (1)
                    //
                    // if sim >= delta, recurse if the child has sufficent words to
                    // be it's own MIU.
                    //

                    if (sim >= delta)
                    {
                        rval = SplitMIUs(tdoc, child, docfreq, stats);
                    }

                    //
                    // apply condition (2)
                    //
                    // if sim < delta && sim >= epsilon, then this
                    // node is the final miu
                    //
                    // currently unimplemented
                    //

                    /*else if (sim < delta && sim >= epsilon)
                    {
                        rval = TRUE;
                    }*/

                    //
                    // apply condition (3)
                    //
                    // if sim < epsilon, then look for an adjacent node
                    //

                    else
                    {
                        // TODO
                        // fix this to do the full adjacent search
                        rval = TRUE;
                    }


                    if (childfreq)
                    {
                        hashmap_del(childfreq);
                    }

                    if (!rval)
                    {
                        break;
                    }
                }
            }
        }

        return (rval);
    }
```

7.2 Proximity Scoring

We implemented the proximity scoring algorithm in the perl script `proxscore.pl`. For the theory behind proximity scoring, see Section 5.2. Like `miuanalysis.exe`, `proxscore.pl` is a command line program that reads an HTML file and outputs a proximity score. While we only tested the script on a Win32 platform, as a generic perl script the program should work on any platform that implements the perl language.

Usage of `proxscore.pl` is completely command line driven and fairly complicated. In the most basic mode, the script reads from a HTML file specified in a command line argument and outputs the final proximity score on `STDOUT`. The program's built in usage summary provides a good starting point.

```
perl proxscore.pl htmlfile query [options]

htmlfile - the html file to retrieve the score for
query - a single quoted parameter with the query words, words should be
        seperated by spaces

options:

Help and Debugging
-h, --help      print this usage information
-v, --verbose   each time this option appears, increase the verbosity
                level of the output

Algorithm Parameters
-c, --cutoff=n              sets the maximum span length to n
-stem, --stemming           forces stemming
-nostem, --nostemming       forces no stemming
-stop, --stoplist=FILE      sets stoplist to FILE,
                            if FILE is "none" then do not use a stoplist
-f, --function=PERLFUNC     sets the option function to the perl function
                            determined by PERLFUNC, i.e. "sub { $_[0] }"
                            for the identify function.

Defaults
 - Does not use a cutoff
 - Uses stemming
 - Uses stoplist.txt for the stoplist, otherwise, no stoplisting
 - Uses f(x) = sqrt(x) as the option function
```

Unlike `miuanalysis.exe`, `proxscore.pl` requires a query to work. The query and the path to an HTML file are the only required options for proximity scoring. The most interesting optional argument is the function. This changes the function used to modify the raw span length. Consequently, Equation 7.1, the equation used in the code which uses $f(|\sigma_i|)$, differs from Equation 5.6.

$$\text{proximity score} = \sum_{i=1}^{\text{total spans}} \frac{1}{f(|\sigma_i|)} \tag{7.1}$$

The default option function is $f(x) = \sqrt{x}$ as noted in the comments. However, the program interprets any perl function provided on the command line. For example, the following command uses the function $f(x) = \log_2(x)$ in Equation 7.1 on the HTML file `myhtmlfile.html` with query *book reviews*.

```
perl proxscore.pl myhtmlfile.html "book reviews"
                  "--function=sub { log($_[0])/log(2) }"
```

One caveat to the `proxscore.pl` script is that it requires three custom modules: `debug`, `stemming`, and `proximity`. These packages are in a different directory, `perl_common`, and the `proxscore.pl` script does not try to dynamically add that directory to the library or package include path.

Using perl, the critical components of the implementation are concise. The main functions in the program are `ProximityScore_SpanBasic` and `Proximity::GetQueryWordSpan` in the `proximity` module. The `ProximityScore_SpanBasic` computes the proximity score using Equation 7.1 from the spans returned by `GetQueryWordSpan`.

We reproduce the two functions `ProximityScore_SpanBasic` and `GetQueryWordSpan` below. The code listing ends on page 55.

```
#
# ProximityScore_SpanBasic
#
# Compute a basic span proximity score.  That is, calculate
#
# total_spans
#    -----
#    \              1
#     )     -----------------
#    /      f(span_length(i))
#    -----
#    i = 0
#
# where total_spans is the total number of spans in the text block and
# a span is a block of text with a unique starting point which contains
# all the words in the query.  The function f(x) is a function used to
# modify the result.
```

```
#
# usage: $score = ProximityScore_SpanBasic($query_aref, $words_aref, [%opts])
#   $query_aref - a reference to the list of words in the query
#   $words_aref - a reference to the list of words in the text
#   $opts - an optional hash containing optional values.  Valid entires are
#               f - a reference to a function to use for modify the result,
#                   this is the identify function if not specified
#               cutoff - the cutoff value to use
#
# returns:
#   $score - a floating point proximity score
#

sub ProximityScore_SpanBasic
{
    my $query_aref = shift @_;
    my @nextwordlist = @{shift @_};

    my %opts = @_;

    my $fref = $opts{"f"};
    if (!defined($fref))
    {
        $fref = sub { sqrt($_[0]) };
    }

    my $score = 0.0;

    while (@nextwordlist)
    {
        my ($nextwordlist_aref, $span_aref) =
        Proximity::GetQueryWordSpan($query_aref,
                                 \@nextwordlist, $opts{"cutoff"});

        @nextwordlist = @{$nextwordlist_aref};
        my @span = @{$span_aref};

        if (@span)
        {
            $score += 1/&$fref(scalar(@span));

            Debug::print_extensive("span: ", join (" ", @span), "\n");
        }
    }

    return ($score);
}

#
# GetQueryWordSpan
#
# retrives the first and shortest span of text that contains all of the query
# words starting at the first word in the input list of words
```

```
#
# usage: ($nextwordlist_aref, $span_aref) =
#           GetQueryWordSpan($query_aref, $wordlist_aref, [$max_cutoff])
#   $query_aref - a reference to the query as an array of words
#   $wordlist_aref - a reference to the words of the text as an array of words
#   $max_cutoff - an optional parameter giving the maximum length of the span
#                  if this is omitted, there is no maximum length
#
# returns:
#   $nextwordlist_aref - a reference to the word list after all words up to
#                        and including the first query word were removed, this
#                        array is empty if no words in the query were found
#   $span_aref - a reference to the words in the span, if not all of the words
#                in the query appeared, or the cutoff was hit, this has
#                length 0
#

sub GetQueryWordSpan
{
    my @query = @{shift @_};
    my @words = @{shift @_};

    my $cutoff = shift @_;
    if (!defined($cutoff))
    {
        $cutoff = POSIX::INT_MAX;
    }

    # define our variables
    my @span;
    my @nextwordlist;
    my $first_word = 0;

    while (scalar(@words) > 0 && scalar(@query) > 0 && $cutoff > 0)
    {
        # get the current word and iterate to the next one
        my $word = shift @words;

        if ($first_word)
        {
            # if we have seen the first word, save the span, and decrement
            # the cutoff value
            push @span, $word;
            $cutoff--;
        }

        # run through all query words to see if any of them occur
        my $qword_index = 0;
        foreach my $qword (@query)
        {
            if ($word eq $qword)
            {
                # make sure we start saving everything from this point, unless
                # it has already been set and saved
                if (!$first_word)
                {
                    $first_word = 1;
                    push @span, $word;

                    # save this pointer
                    @nextwordlist = @words;
                }

                # remove this word from the query
                splice (@query, $qword_index, 1);
            }
            $qword_index++;
        }
    }

    # test and see why we terminated
    if (scalar(@query) > 0)
    {
        # here, we didn't see all the terms, so don't return anything in
        # the span
        splice (@span);
    }

    return (\@nextwordlist, \@span);
}
```

7.3 Query Retrieval

There are two steps to retrieving the pages for a query. The first step is retrieving the data from the algorithmic search engine. In our implementation, we use the publicly available Google SOAP interface. We describe this program in Section 7.3.1. The second step is retrieving and locally caching images and other server-stored material. See Section 7.3.2 for our implementation.

Warning While all the programs involved in query retrieval are implemented in perl, we may have used directory separators specific to the Win32 platform. Please check this problem before running these scripts on any non Win32 platform.

The source code to perform query retrieval is in the `soap_queries` directory.

7.3.1 Google SOAP

Our program to retrieve queries is `soap_queries.pl`. This program requires a query from the command line and optionally, the number of pages to retrieve. The first sample command retrieves the top 200 pages for the query "book reviews" and the second sample command retrieves the top 140 pages for the query "book reviews."

```
perl soap_queries.pl "book reviews"
perl soap_queries.pl "book reviews" 140
```

For the rest of this section, "book reviews" is the example query.

The first thing `soap_queries.pl` does is to create a subdirectory of `..\Query_Results\` with the query name. In our example, it creates `..\Query_Results\bookreviews\`. The program then begins retrieving pages from Google using their SOAP web service interface. The results directory is determined by the program variable `$results_dir`.

The program next creates a file with the query name in that directory. For the example query, it will create the file `..\Query_Results\book reviews\book reviews.txt`. Eventually, this file will contain the list of local pages and their original URLs. Each line of this file contains two tab-delimited entries, local page name and original URL, in the following format.

```
local_page_name URL
```

One quirk about the Google SOAP interface is that a single query will return no more than 10 results. Consequently, to get 200 results, the program must perform 20 queries. The program controls the total number of pages it retrieves with the variable `$max_cached_results`.

Our algorithm to retrieve the pages only counts a page if Google has a cached copy of the page. We used this approach to minimize the potential difference between the web page as indexed by Google and the live version of the page. If Google has a cached copy of the page, the algorithm retrieves the page, strips Google's header, and stores it in the file `page%i.html` where `%i` is the current zero-indexed result number. So, for the 5th page on the query "book reviews", the algorithm would place the HTML for that page in `..\Query_Results\book reviews\page4.html`. After locally storing the HTML, `soap_queries.pl` calls our image caching script `cache_images.pl`. The program repeats this process for each of the pages retrieved.

7.3.2 Image Caching

Our program for locally caching images is `cache_images.pl`. The program has two required options. The first is the local HTML file name, the second is the original URL of the page. In the sample command, `cache_images.pl` will cache `page10.html` which had the original URL `http://www.yahoo.com`.

```
perl cache_images.pl ".\query\page10.html" "http://www.yahoo.com"
```

The first step of the algorithm is to create a directory for locally cached images. Our program creates a name for this directory by appending `_cache` to the base form of the file name. For our example, the program would create the directory `.\query\page10_cache\`.

Next, the script parses the HTML using the `HTML::Transform` library [NF-HTML]. Using this library, we specify a perl function to call for each occurrence of a specific HTML tag in the file. We created handlers for the tags <img>, <image>, <link>, and <base>. The `base` function, called when the library encounters a <base> tag, stores the value of the `href` attribute to replace the URL passed on the command line. It then removes the `href` attribute from the <base> tag.

For <img>, <image>, and <link>, we simply store the file referenced by the `href` attribute in the cache directory and update the `href` attribute to point to the locally cached file.

The program stores the result of this transformation in a file. In our example, the file is `.\query\page10_cache.html`. In general, the file name is formed by appending the string `_cache` to the base file name.

While this procedure works successfully on many web pages, we encountered two problems. First, many web pages store content in other tags. Other web pages store image references in JavaScript. Handling all of these cases was beyond the scope of this program. In order to cache these pages `cache_images.html` stores another version of the file with a <base> tag pointing toward the original URL. This file is called `.\query\page10_base.html`.

The second problem is that the `HTML::Transform` library did not robustly handle all pages. HTML is not a rigid specification. However, the failures in the `HTML::Transform` library were caused by tag balance problems. To fix this behavior, we removed the code for tag balancing from the `HTML::Transform` library. The changes are noted in the `Transform.pm` file in the `soap_queries` directory.

7.4 Re-ranking

As described in Section 4.1.2, we have two algorithms for re-ranking. The code for our implementation of these algorithms is in the `rerank` directory. In this section we only present a usage guide to these programs. Please see the CD accompanying this report for the full source code listing. With the description of the algorithms from Section 4.1.2 and the comments in the code, the code is easy to understand.

These programs are portable to any platform that implements perl.

7.4.1 MIUs

Our program for MIU re-ranking is `rerank_miu.pl` which implements the theory described in Section 4.1.2.1. The program requires three options: a directory full of web pages, a query, and a type of MIU analysis.

```
perl rerank_miu.pl webpages_dir query td|topdown|bu|bottomup [options]

webpages_dir - The directory with web pages.  This directory needs to
               contain files of the form page#.html, where # is a number.
               Also, it needs to contain a file named "query.txt" where
               query is the option below which contains page to url
               mappings.
query - a single quoted parameter with the query words, words should be
        separated by spaces
type - either td, topdown for top-down analysis or bu, bottomup for
       bottom-up MIU analysis
```

```
options:

Help and Debugging
-h, --help      print this usage information
-v, --verbose   each time this option appears, increase the verbosity
                level of the output

Parameters
-p, --prog=STRING    use STRING as the command for miu analysis program
-a, --args=STRING    give STRING as the command line arguments to the
                     miu analysis program (prog)
-c, --cache          use cached MIUs if available, creates a cache otherwise
-noc, --nocache      force creation of new MIUs
-l, --log            log the miuanalysis results for each file
-nol, --nolog        do not log the miuanalysis results for each file

Defaults:
- Uses "miuanalysis.exe" the command for miu analysis
- Uses no additional command line arguments
- Uses cached MIUs if available, otherwise, creates them with names
  like page01.td.miu (for top down miuanalysis) in the webpages_dir
  directory.  The files are assumed to be in the correct format.
- Uses logging
```

The usage description explains all the options. The command lines used for our bottom up and top down MIU re-ranking follow (see `rerankmius_bu.bat` and `rerankmius_td.bat`). The batch file interpreter replaces the `%query%` token with the actual query.

```
perl rerank_miu.pl "..\Query_Results\%query%" "%query%" bu
       -a "-cfg bottomup.cfg"
perl rerank_miu.pl "..\Query_Results\%query%" "%query%" td
       -a "-cfg topdown.cfg"
```

In the first case, we perform bottom up re-ranking using caching and logging using `miuanalysis.exe` (the default) and the command line argument `-cfg bottomup.cfg` directs `miuanalysis.exe` to use the `bottomup.cfg` configuration file for `miuanalysis.exe` (see Section 7.1.1.1). In the second case, we perform top down MIU analysis with the same options but using the `topdown.cfg` configuration file instead. Using these command templates, we re-rank our "book reviews" using bottom up MIU analysis with the following command.

```
perl rerank_miu.pl "..\Query_Results\book reviews" "book reviews" bu
       -a "-cfg bottomup.cfg"
```

The `rerank_miu.pl` script writes its output to `STDOUT`. The output of the program is a line for each page. Each line contains four tab (`\t`) delimited entries.

```
rank local_html_file miu_window_size url
```

- `rank` is the zero-indexed rank of `local_html_file`.
- `local_html_file` is the name of the local page, e.g. `page5.html`
- `miu_window_size` is the number of MIUs required to contain all the query terms.
- `url` is the original URL of the file.

7.4.2 Proximity Scoring

Our program for proximity score re-ranking is `rerank_prox.pl` which implements the theory described in Section 4.1.2.2. The program requires two options, a directory full of web pages and a query.

```
perl rerank_prox.pl webpages_dir query [options]

webpages_dir - The directory with web pages.  This directory needs to
               contain files of the form page#.html, where # is a number.
               Also, it needs to contain a file named "query.txt" where
               query is the option below which contains page to url
               mappings.
query - a single quoted parameter with the query words, words should be
        seperated by spaces

options:

Help and Debugging
-h, --help      print this usage information
-v, --verbose each time this option appears, increase the verbosity
                level of the output

Parameters
-b, --bins=n         use n bins for reranking.
-p, --prog=STRING    use STRING as the command for proximity scoring
                           program
-a, --args=STRING    give STRING as the command line arguments to the
                           proximity scoring program (prog)

Defaults:
- Uses "perl proxscore.pl" as the program for proximity scoring
- Uses no additional command line arguments
- Uses bins = number of query words
```

The usage description explains all the options. The command lines used for our proximity score re-ranking follows (see `rerankprox.bat`). The batch file interpreter replaces the `%query%` token with the actual query.

```
perl rerank_prox.pl "..\Query_Results\%query%" "%query%"
       -a "--stoplist=none" --bins=40
```

This sample command runs proximity score re-ranking with no stoplist and 40 bins. To re-rank the query "book reviews" then, we would use the following command.

```
perl rerank_prox.pl "..\Query_Results\book reviews" "book reviews"
       -a "--stoplist=none" --bins=40
```

The `rerank_prox.pl` script writes its output to `STDOUT`. The output of the program is a line for each page. Each line contains four tab (`\t`) delimited entries.

```
rank local_html_file proximity_score url
```

- `rank` is the zero-indexed rank of `local_html_file`.
- `local_html_file` is the name of the local page, e.g. `page5.html`
- `proximity_score` is the proximity score of `local_html_file` for the query.
- `url` is the original URL of the file.

7.5 Human Rating

The programs discussed in this section are written in C# and compiled for the .NET platform using Visual Studio .NET. They are very rough programs designed solely for one task and are not very adaptable.

Both of the programs have directories and queries encoded within the source code of the program – there is no way to change these options without recompiling the program.

In this section, we simply present and explain the user interface of each program. See the accompanying CD for the source code listings in the `RatingApp` and the `RatingAppOrder` directories.

7.5.1 Precision Scoring

Our first program `RatingApp.exe` performs precision scoring. See Figure 6.3 for a screenshot of the full `RatingApp.exe` window and Figure 7.2 for just the user interface.

The program reads a list of web pages to display for precision scoring from the directory `L:\Human_Results\%query%\order-relevance.txt` where `%query%` is the current query, found at the end of the title bar in Figure 7.2. In order for precision scoring to work, the directory `L:\Query_Results\%query%\%query%.txt` must contain a simple map between pages and URLs as described on page 56.

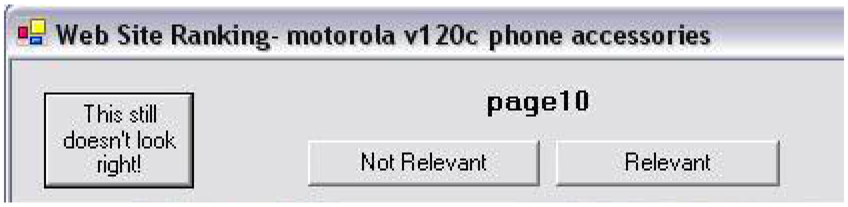

Web Site Ranking- motorola v120c phone accessories
This still doesn't look right!
page10
Not Relevant
Relevant

Figure 7.2: The user interface for our precision scoring application.

As soon as a user clicks the "Not Relevant" or "Relevant" button, the program presents the next page. This continues until the user has rated all pages in `L:\Human_Results\%query%\order-relevance.txt`.

If, for some reason, the original page does not display properly, the user can click the "This doesn't look right!" button. The first time a user presses this query, the program attempts to load a different cached version of the page. The second page the interface shows is the `base` page created by `cache_images.pl` described in Section 7.3.2. After the user presses the button the first time, the program changes its name to "This still doesn't look right!" If the user presses the button a second time, the program loads a live view of the page.

The program writes the results of the comparisons to the file `L:\Human_Results\%query%\%user%.rel.txt` where `%user%` is the user identifier for the current user. Each line in the file contains the word "page", a tab, the number of the page, a tab, and the result, "0" or "1." A result of "0" indicates the user judged the page not relevant, whereas a result of "1" indicates the user judged the page relevant. In the following example, `page5.html` and `page40.html` are relevant to the query, but `page78.html` and `page1.html` are not.

```
page   5    1
page   78   0
page   40   1
page   1    0
```

7.5.2 Comparisons

Our second program `RatingAppOrder.exe` performs relevance ordering. See Figure 6.4 for a screenshot of the full `RatingAppOrder.exe` window and Figure 7.2 for just the user interface.

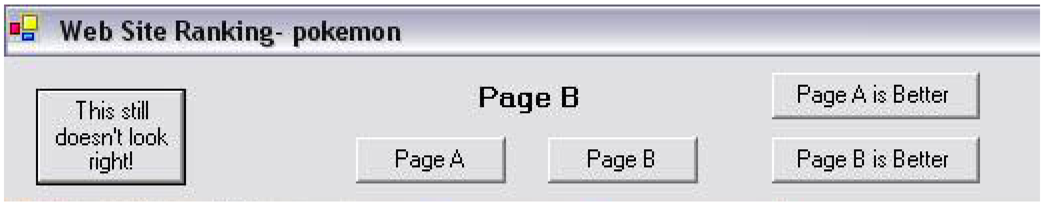


Figure 7.3: The user interface for our relevance ordering application.

The program reads a list of web pages to display for relevance ordering from the directory `L:\Human_Results\%query%\order-comp.txt` where `%query%` is the current query, found at the end of the title bar in Figure 7.2. In order for relevance ordering to work, the directory `L:\Query_Results\%query%\%query%.txt` must contain a simple map between pages and URLs as described on page 56.

In the interface of `RatingAppOrder.exe` the user chooses which of two pages is more relevant to the query. They switch between pages by pressing the "Page A" or "Page B" buttons. The current page is shown in the label that currently displays "Page B." After the user makes his or her judgment about the pages, he or she presses the "Page A is Better" or "Page B is Better" as applicable. After the user presses this button, the program displays the next set of pages for comparison, until it has exhausted the list of comparisons from `order-comp.txt`.

If, for some reason, the original page does not display properly, the user can click the "This doesn't look right!" button. The first time a user presses this query, the program attempts to load a different cached version of the page. The second page the interface shows is the `base` page created by `cache_images.pl` described in Section 7.3.2. After the user presses the button the first time, the program changes its name to "This still doesn't look right!" If the user presses the button a second time, the program loads a live view of the page.

The program writes the results of the comparisons to the file `L:\Human_Results\%query%\%user%.comp.txt` where `%user%` is the user identifier for the current user. Each line in the file contains the local HTML file name for page A and page B and the result, either "A" or "B." If the result is "A", then the user ranked the first page (page A) better than the second page (page B). Likewise, if the result is "B", then the user ranked the second page (page B) better than the first page (page A). In the following example, a user compared three sets of pages and rated `page7.html` above `page161.html`, `page27.html` above `page3.html`, and `page8.html` above `page1.html`.

```
page161.html  page7.html  B
page27.html   page3.html  A
page1.html    page8.html  B
```

7.6 Other Scripts

We wrote many scripts to aid our project over the course of two semesters. Unfortunately, we cannot fully explain all of them. Table 7.2 contains a list of scripts and a brief description of each script. The scripts are in the `misc` directory on the CD accompanying this report.

Script Name	Description
`check_caching.pl`	Checks the image caching of pages from `cache_images.pl`. The output from this script is a list of uncached pages and directories.
`create_human_directories.pl`	Creates the `human_results` directory structure for all the queries inside the script.

continued on next page

Script Name	**Description**
`create_kendall_input.pl`	Outputs a matlab .m file on `STDOUT` with the data to run Kendall's Tau comparisons on the four rankings: Google's, bottom up MIU analysis, top down MIU analysis, and proximity scoring.
`direct_matrix_gen.pl`	Generates the matrix of comparisons from the human comparison results, and outputs a matlab .m file on `STDOUT` with this matrix. The script reads the human comparisons from `STDIN`.
`fix_uncached_pages.pl`	Reads the input from `check_caching.pl` on `STDIN` and re-runs the `cache_images.pl` script on the uncached pages.
`lookup_page.pl`	Determines the re-ranking of a page (e.g. `page5.html`) for a given query and analysis method (e.g. bottom up).
`query_page_set.pl`	Outputs the top 10 pages from each of the three re-ranked sets of pages in one combined list of local HTML files. We used the output of this script for as input for `randomize_exhaustive_comparisons.pl`.
`randomize_comparisons.pl`	Outputs a certain number of random comparisons within a fixed size set.
`randomize_exhaustive_comparisons.pl`	Reads a list of pages from `STDIN` and outputs a randomized list for an exhaustive comparison among the pages. Writes this list to `STDOUT`.
`randomize_relevance.pl`	Reads `order-relevance.txt` for a list of pages in the `human_results` directory, randomizes this list, and then overwrites the original `order-relevance.txt` file with the new list.
`rerank_query.bat`	A batch file to run all three re-ranking methods for a query.
`rerankmius_bu.bat`	A batch file to get the bottom up re-ranking for a query.
`rerankmius_td.bat`	A batch file to get the top down re-ranking for a query.
`rerankprox.bat`	A batch file to get the precision scoring re-ranking for a query.

continued on next page

Script Name	Description
`select_10.pl`	Outputs the top 10 pages from each of the three re-ranked sets of pages in three lists.

Table 7.2: Brief descriptions of the miscellaneous scripts used in our project.

7.7 Algorithmic Analysis

In this section, we present the algorithmic complexity of our three analysis methods: bottom up MIU analysis, top down MIU analysis, and proximity scoring. Also, we give the runtime of our re-ranking algorithms. We do not present formal proofs of the stated complexities. However, we justify the run-time values using the theoretical algorithms.

Bottom Up MIU Analysis

In our implementation of the bottom up algorithm, after the algorithm merges two nodes, it must reexamine all pairs of nodes for potential header-content merging. In the case of a linear tree, its runtime is quadratic: $\mathcal{O}(n^2)$. On average, it will be much less than this. n is the number of nodes in the tag tree.

In this analysis, we assume that constant time, $\mathcal{O}(1)$, implementations of `MIU_GetText` and `MIU_MergeAdjacent` are possible – this is not the case in the current code. We also do not consider the complexity of constructing the HTML DOM tree.

Top Down MIU Analysis

In the process of splitting and examining the tree, the algorithm can only look at each node once. Consequently, the big-O runtime of the top down MIU analysis is linear: $\mathcal{O}(n)$. Again, n is the number of nodes in the tag tree.

In this analysis, we assume that constant time, $\mathcal{O}(1)$, implementations of `MIU_GetText` and `MIU_MergeAdjacent` are possible – this is not the case in the current code. We also do not consider the complexity of constructing the HTML DOM tree.

Proximity Scoring

The big-O runtime of proximity scoring is quadratic: $\mathcal{O}(n^2)$. However, with a cutoff value, the big-O runtime is linear: $\mathcal{O}(n)$. In these cases, n is the number of words in the file. With no cutoff value, the algorithm might have to search until the end of the file for every occurrence of a query term. If the file has n words, this could take n steps for each of n possible term matches, demonstrating a quadratic run-time.

With a cutoff value, there is a constant number of terms the algorithm will search for each query term that occurs in the text. Thus, if the file has n words, this algorithm would only take a constant number of steps for each of n possible term matches, demonstrating a linear run-time.

We hypothesize that there is a purely linear algorithm for proximity scoring, even without using a cutoff.

MIU Re-Ranking

Since this algorithm involves a similar process to "proximity scoring" the run-time analysis is the same. The MIU window re-ranking algorithm has a $\mathcal{O}(n^2)$ runtime to compute the MIU windows. The other components of re-ranking only involve sorting. Presumably, the algorithmic rank lookup is $\mathcal{O}(1)$ which produces a final runtime of $n\log(n) + nm^2$, where n is the number of pages and m is the maximum number of MIUs on a page.

Proximity Score Re-Reranking

Proximity score re-ranking involves a sort, a division into sets, and another sort. The complexity of these steps is $n\log(n)$, n, and $n\log(n)$ respectively. Consequently, the overall complexity of the proximity score re-ranking algorithm is $\mathcal{O}(n\log(n))$.

Chapter 8

Our Results

We use two different metrics to measure the accuracy of our re-rankings. The first metric, *precision* (see Section 4.2.1), determines whether a web page is relevant or not relevant to a given query. The second metric, *relevance ordering* (see Section 4.2.2), determines how “well ordered” the results from each re-ranking are compared to a standard ranking determined by humans. Both metrics of re-ranking accuracy are important in understanding our results. Take for example, the query “guitar hanger stand.” The top ten Google pages for this query received a 90% precision rating, that is nine of the ten pages were deemed relevant, by humans, to the query.

However, the ranking for this query, generated by humans, tells a different story. The top ten pages, as ranked by humans, include only three of Google’s top ten pages (see Table 8.1). Thus, only using precision, or only using relevance ordering would produce an incomplete picture.

Human and Google Relevance Orderings for “guitar hanger stand”

Position	Google’s Ranking
1	`page66.html`
2	`page47.html`
3	`page13.html`
4	`page58.html`
5	`page16.html`
6	`page25.html`
7	`page17.html`
8	`page4.html`
9	`page6.html`
10	`page5.html`

Table 8.1: Human and Google Relevance Ordering for the query “guitar hanger stand”. The left column shows the human ranking positions and the right column shows the Google ranking of the page in that position. Thus, the first row reads, humans felt the most “important” page to the query “guitar hanger stand” was the page Google ranked 66th using PageRank.

The remainder of this chapter discusses the results of our project. In terms of precision, the bottom up method performed the best. In terms of relevance ordering, Google performed the best. Interestingly, the proximity scoring method performed well when Google has poor relevance ordering performance. However, neither of our two metrics, precision and relevance ordering, shows that any of our methods have a statistically significant advantage over the algorithmic search engine we used for testing, Google.

In addition to discussing the formal results we found, we provide a discussion of interesting observations about our data. These observations were obtained during data analysis, and though they have not been thoroughly explored, they are important to note.

8.1 Metric 1: Precision Results

Table 8.2 below illustrates all our results for precision. A summary of our results follows in Table 8.3. These data shows that the bottom up approach has the highest average precision, and that it has precision strictly greater than Google the most often (27.6% of the time). The bottom up approach also has the highest precision the most often (70.7% of all queries measured, including ties), that is, on 41 of 58 queries the precision of the bottom up method was greater than or equal to all the other methods. Our bottom up method, as well, achieves perfect precision the most often (36.2%), tied with the top down approach. However, an F-Test on the data failed to show any between group differences within a 95% confidence interval; $P = 0.29$.

	BU	**TD**	**Prox**	**Google**
a summary on world war one	0.9	0.4	0.5	0.9
always on top utility	0.7	0.7	0.2	0.7
bartender drinks mixed drink recipes	1.0	1.0	1.0	0.9
blue mountain	1.0	1.0	0.9	1.0
boston apartments	0.8	0.8	0.8	0.8
brand new rock	0.6	0.4	0.5	0.7
certification in housing	0.8	0.8	0.7	0.9
cheap tech gear	0.6	0.6	0.1	0.7
church service	0.6	0.6	0.5	0.6
colleges that offer training for private investigators	0.5	0.2	0.4	0.6
contour pillow	1.0	1.0	1.0	1.0
cybernet ventures	0.9	0.9	0.9	0.9
eddie bauer infant car seat	1.0	1.0	1.0	1.0
elva corporation	0.0	0.1	0.2	0.0
fleet space museum calif	0.4	0.0	0.2	0.1
forklift operator	0.5	0.5	0.5	0.5
free chess tactics	0.4	0.7	0.8	0.2
front bumper	0.8	0.8	0.9	0.8
gender differences in education	0.9	0.9	0.8	0.8
guitar hanger stand	0.9	0.9	0.9	0.9

continued on next page

	BU	**TD**	**Prox**	**Google**
harry potter	0.9	0.9	0.8	0.9
health insurance companies of tampa bay florida	0.3	0.0	0.2	0.0
health insurance probability and accountability act	0.8	0.6	0.6	0.9
infectious diseases	1.0	1.0	1.0	1.0
java	0.7	0.7	1.0	0.7
john milton	1.0	1.0	0.6	1.0
jpeg cd burner slide show dvd player	0.8	0.9	0.7	0.8
kingdom hearts 59 puppies	0.7	0.4	0	0.7
laptop cases	0.8	0.8	0.8	0.8
learn thai massage in thailand	1.0	1.0	0.6	0.9
los molinos	1.0	1.0	0.8	1.0
lyrics eminem super man	0.7	0.6	0.4	0.8
manual transmission near phase	0.1	0.0	0.2	0.3
maternity scrubs	1.0	1.0	0.9	1.0
mindstorm lego robotics	0.9	0.9	1.0	0.8
motorola v120c phone accessories	1.0	1.0	1.0	1.0
mystery party	1.0	1.0	0.9	1.0
nt administration helpdesk	0.7	0.6	0.4	0.7
paint stick new inventions	0.1	0.0	0.0	0.1
pokemon	1.0	1.0	1.0	0.9
pretty sunset landscapes	0.8	0.3	0.5	0.7
san luis obispo	1.0	1.0	0.8	1.0
sebring headlight	1.0	1.0	0.9	1.0
sesame street cookie monster	0.9	0.9	0.9	1.0
she loves me and veronica	0.3	0.2	0.6	0.3
sick building syndrome	1.0	1.0	1.0	1.0
star trekker	0.9	0.9	0.6	0.9
statistical language analysis	0.6	0.6	0.7	0.4
structural design context applications	0.4	0.5	0.3	0.5
sun city arizona tours to laughlin and vegas	0.8	0.7	0.8	0.6
super child model	0.7	0.5	0.5	0.5
terrorism iraq afghanistan	1.0	1.0	1.0	0.7
the actors gang	1.0	1.0	0.6	0.9
the psychological corporation	1.0	1.0	1.0	0.9
thomas the tank boco	1.0	1.0	0.9	1.0
u.s.a olympic ice skating contenders	0.8	0.8	0.4	0.8
un members	1.0	1.0	1.0	0.9
world junior hockey	1.0	1.0	1.0	1.0
average	0.78	0.73	0.68	0.75

Table 8.2: Here, BU stands for our bottom up method, TD is our top down method, and Prox refers our proximity scoring method. This table gives the precision scores of each of our three methods and Google for all fifty-eight queries that were tested. The precision scores are between 0 and 1. For instance, the 0.7 in the Prox column for the query *statistical language analysis* means that of the top ten pages returned by our proximity scoring algorithm, the team judged seven pages to be relevant to the query.

	Average Precision	Queries with Better Precision than Google	Queries with Best Precision
Google	0.75	0	37
Bottom Up	0.78	16	41
Top Down	0.73	14	34
Proximity Scoring	0.68	15	24

Table 8.3: This table is a summary of the precision scores of our three methods and Google. Again, each precision score is between 0 and 1. The second column measures how many queries the method has strictly greater precision than Google. The third column counts the queries where the precision was the best, or equal to the best, of any of the methods.

8.2 Metric 2: Relevance Ordering Results

The table below illustrates all our results for relevance ordering. These data show that Google has the highest average relevance ordering, .1210. This is followed closely by the bottom up approach, with an average relevance ordering of .1069. Again, however, an F-Test on the data failed to show any between group differences within a 95% confidence interval; $P = 0.34$. Although proximity scoring had the lowest average relevance ordering, its results were the most interesting. On all but 7 of the 29 queries tested, proximity scoring achieved a positive correlation score when Google achieved a negative correlation score, and vice versa. Proximity scoring achieved a negative relevance ordering 13 times, and Google 9 times and one score exactly zero. This suggests that Google rates web pages with a significantly differently method than just proximity scoring. In contrast, top down and bottom up achieved the *exactly* the same relevance ordering as Google 8 times out of 29. This suggests the same argument that our statistical tests show: that our methods did not re-rank the results differently enough for there to be any variation between the methods.

	BU	**TD**	**Prox**	**Google**
a summary on world war one	0.5714	-0.0197	-0.2857	0.7192
bartender drinks mixed drink recipes	0.3123	0.1383	-0.1779	0.3518
brand new rock	-0.0467	0.1333	0.1333	-0.1800
cheap tech gear	0.0646	0.0400	-0.1631	0.2369
colleges that offer training for private investigators	0.2328	0.1164	0.2751	0.3016
contour pillow	0.1813	0.1813	-0.2164	0.1813
cybernet ventures	0.5789	0.5789	-0.2749	0.4386
eddie bauer infant car seat	0.1368	0.1368	-0.4105	0.2632
free chess tactics	0.0997	0.0256	-0.0256	0.151
guitar hanger stand	-0.1082	0.0736	0.2121	-0.2814
harry potter	-0.0526	-0.0526	0.2164	-0.0526
infectious diseases	0.000	0.000	0.1684	0.000

continued on next page

	BU	**TD**	**Prox**	**Google**
java	-0.1895	-0.1895	0.4842	-0.1895
john milton	0.1905	0.1905	-0.2381	0.181
laptop cases	0.3789	0.3789	-0.200	0.3789
learn thai massage in thailand	0.1169	0.0909	0.1515	0.1342
los molinos	-0.1912	-0.1912	0.3824	-0.1912
maternity scrubs	0.0441	-0.0147	0.0294	0.0441
motorola v120c phone accessories	-0.1600	0.1333	0.1067	-0.0867
mystery party	0.2526	0.2526	-0.1368	0.2526
paint stick new inventions	-0.1206	0.0603	0.3429	-0.1365
pretty sunset landscapes	0.1111	-0.1058	0.1481	0.0952
sesame street cookie monster	-0.3400	-0.3867	0.08	0.0333
sick building syndrome	-0.0105	-0.0105	0.2105	-0.0105
statistical language analysis	0.2526	0.2421	-0.1053	0.1579
terrorism iraq afghanistan	0.2933	0.2533	0.2067	0.1667
the psychological corporation	0.2000	0.2000	0.1053	0.0211
thomas the tank boco	0.3676	0.3043	-0.3360	0.4941
u.s.a olympic ice skating contenders	-0.1474	-0.1474	0.1789	-0.0316
un members	0.1895	0.1895	-0.1158	0.1895
average	0.1069	0.0867	0.0249	0.1211

Table 8.4: Again, BU stands for our bottom up method, TD is our top down method, and Prox refers our proximity scoring method. This is a list of the relevance scores for all thirty of the queries. Each score is between -1 and 1.

8.3 Interesting Observations About Our Results

Here are some observations on the results we obtained. We believe these observations would be fruitful to explore in future work.

We discuss the results of both precision (see Section 4.2.1) and relevance ordering (see Section 4.2.2).

Please note that in this section we use the names of our methods to describe the use of our methods *and* the re-ranking used with that method. For example, we use the term "bottom up" to describe the process of bottom up MIU analysis together with the subsequent re-ranking using the results from bottom up MIU analysis.

8.3.1 Observations on Relevance Ordering Data

Our first observation on the relevance ordering data is that proximity scoring consistently (98.3% of the time) shifts over half of Google's top ten pages out of the top ten ranking positions, and in many instances (44.8% of the time) all of Google's top ten pages were shifted out of the top ten position. This indicates that Google's ranking criteria differ significantly from our proximity scoring re-ranking criteria.

Another interesting observation on the relevance ordering data is that only one query re-ranked with MIU analysis had an MIU window size (see Section 4.1.2.1 for details on MIU windowing) greater than one in the top ten results. This means that in almost all of the top ten pages, our MIU analysis algorithms constructed one MIU with all the query terms. The one exception was the query "sun city arizona tours to laughlin and vegas" using the bottom up MIU analysis method. We are unsure what the significance of this observation is; however, we felt it note worthy.

Our third interesting observation on the relevance ordering data is that, for each query, proximity scoring shifted at least one very low ranked page (94.8% of the time greater than 100) up to a position in the top 10. Thus, proximity scoring shows a dramatic shift in the pages it finds more important to a query. Additionally, the lowest ranked page in the top ten usually fell at position 5. This can be explained by the binning technique used to re-rank pages based on proximity scoring. Recall from Section 4.1.2.2 that the pages are sorted into bins of size b based on their proximity score, and then sorted within the bin using the ranking from the algorithmic search engine. In our implementation (see Section 7.4), $b = 6$. Thus, each bin held six pages, and within each bin the pages were further sorted by Google's ranking. To show what happens, let us take the example of the query "a summary of world war one." Table 8.5 shows the proximity score ranking of the top six pages for the query

Proximity Score Ranking for "a summary of world war one"

Position	Google's Ranking	Proximity Score
1	`page64.html`	10.1558
2	`page80.html`	6.4675
3	`page81.html`	7.0724
4	`page94.html`	7.3729
5	`page115.html`	8.2716
6	`page194.html`	5.1197

Table 8.5: Proximity score ranking for query "a summary of world war one." This table shows the top six pages (i.e. the pages in the first bin) based on proximity scoring. The first line in the table means: The proximity scoring method re-ranked Google's page 64, with a proximity score of 10.1558, to position 1.

"a summary of world war one." The important feature to note here is that while the pages under Google's Ranking are in ascending order, the proximity scores are not. The pages presented in Table 8.5 are the pages with the top six proximity scores, but they have been reordered according to Google's ranking. Since the bin is of size six, the lowest ranked page will be in position 5. The same phenomena will occur for all subsequent bins, each also of size six.

8.3.2 Observations on Precision Data

Recall from Section 6.2.1 that precision data was collected on the top ten pages returned by each method. With this information we evaluated how precise each method was for each of the 58 queries we tested.

There were quite a few interesting observations made on the precision data. First, it was observed that each of the two times that Google had a precision rating of zero on a query, one of our MIU methods also had a precision rating of zero. In one instance it was the bottom up method, in the other it was the top down method. Second, we noted that each method (Google, proximity scoring, bottom up, and top down) had a zero precision rating on at least one of the queries. This shows that each of the methods will have trouble identifying relevant pages to at least one type of query.

Next, we examined each method for how many times it showed perfect precision (a precision score of 1) on a query (see Table 8.6 for results). A query has a precision score of 1 when the top ten results returned were *all* judged by humans to be relevant to the given query . We found that the two MIU methods (bottom up and top down) had perfect precision more often than the other two methods.

Measurements of Perfect Precision

Method	Number of times with Perfect Precision	Percentage of time with Perfect Precision
Bottom Up	21	36.21%
Top Down	21	36.21%
Proximity Scoring	13	22.41%
Google	15	25.86%

Table 8.6: Number of times each method had perfect precision (precision score of 1) on a query. The first line of the table reads: The bottom up method had perfect precision on 21 of the queries, which is 36.21% of the total queries.

After looking at perfect precision measures, we examined how many times all three of our methods (bottom up, top down, and proximity scoring) had higher precision scores than Google, and how many times Google had a precision score higher than the precision scores for all three of our methods. We found that the count was approximately equal. On eight of the queries (or 13.8%) all of our three methods had higher precisions scores than Google. On seven of the queries (or 12.1%) Google's precision score was higher than the precision scores or each of our three methods. Lastly, on 11 of the queries (or 10.9%) all the methods (bottom up, top down, proximity scoring, and Google) had the same precision score. This data is summarized in Table 8.7.

Relation of Precision Scores

Relation of Precision Scores	Number of queries	Percentage of queries
BU, TD, PS $>$ Google	8	13.8%
BU, TD, PS $=$ Google	11	19.0%
BU, TD, PS $<$ Google	7	12.1%

Table 8.7: Relation of precision scores of each of the four methods (bottom up or BU, top down or TD, proximity scoring or PS, and Google). The first line of the the table reads: bottom up, top down and proximity scoring had higher precision scores than Google on 8 of the queries (or 13.8%).

Chapter 9

Conclusion

In this chapter we discuss our conclusions on how relevance ordering in web searches can be improved.

Our project was to research how to improve relevance ordering in web search. We first explored the Overture Summer Clinic research, and decided to continue the idea of MIU (micro information unit) analysis. We implemented three methods: an adaptation of Li et al's bottom up approach to MIU analysis, an adaptation of Chakrabarti et al's top down MIU approach, and proximity scoring. Both the bottom up and top approaches to MIU analysis exploit the HTML tag structure of the web page. The bottom up method segments web pages into MIUs by recursively attempting to merge the lowest tag nodes together according to the similarity of the text. The top down method employs the opposite approach, recursively considering only the top children nodes, and using a text similarity function to split the page into MIUs. Our third method assigns scores to each web page according to a proximity scoring function. We then re-ranked the top 200 web pages returned by Google for each of 58 queries, using each of the two MIU segmentations and proximity scoring. We used two metrics to measure the effectiveness of our re-rankings: *precision* and *relevance ordering*. Precision is the fraction of how many web pages are deemed relevant to a query out of those top N pages returned by one of our methods. Relevance ordering is a measure of how well correlated the re-ranked results from our three methods are to an ideal human ranking.

After implementing the methods and running testing protocols, we found that on average, the bottom up MIU approach had the highest precision of the four methods: bottom up , top down, proximity scoring, and Google. The bottom up approach also had higher precision than Google's results the most times (16 out of 58 queries). It also tied with Google for the most queries with the best precision (21 out of 58). However, the results were not statistically significant. Google achieved the highest average relevance ordering out of all four methods. Proximity scoring beat Google the most times out of our three methods with respect to relevance ordering, and it seems that when Google had low relevance ordering, proximity scoring often achieved high relevance ordering. However, the average relevance ordering for proximity scoring was the lowest because proximity scoring seemed to do either very well or very poorly.

9.1 Limitations of Our Methods

The three methods we developed are each advantageous in web searching. However, each method has its drawbacks. This section reviews some limitations of each of our methods for improving relevance ordering in web searching.

9.1.1 Limitations of Top Down MIU Analysis

We noticed two limitations to the top down method:

1. Given our implementation of top down, a page is very unlikely to have only one MIU after MIU analysis.

2. Certain pages, because of their structure and our implementation, can only be broken into very few or very many MIUs.

The way top down is implemented, in each iteration of the algorithm, either a parent node must be split into all of its children nodes or the algorithm must come to a halt. The HTML DOM tree (see Section 3.2) for each page uses the <html> tag as the root node. Thus, the children of the root node are the <title> and <body> tags. Recall from Section 5.1.3 that if the content of the children of a parent node is sufficiently similar to the document as a whole, then the parent node is split into its children. In the case of the root node for each tree, the <body> node will be sufficiently similar to the entire document because it is most of the document; the title is only a small portion of the entire document. Thus, though it is possible for the top down algorithm to produce only one MIU, it is highly unlikely based on the structure of the HTML DOM trees and the implementation of the top down MIU analysis approach.

The second limitation of the top down method occurs on a certain type of tree (see Figure 9.1). In this type of tree a majority of the nodes of the DOM tree are the children of a single parent node. Thus, the parent node, which contains the majority of the nodes of the tree as its children, will either remain as one MIU or be split into all of its children. Thus, the web page will either contain just a few MIUs or many, many MIUs.

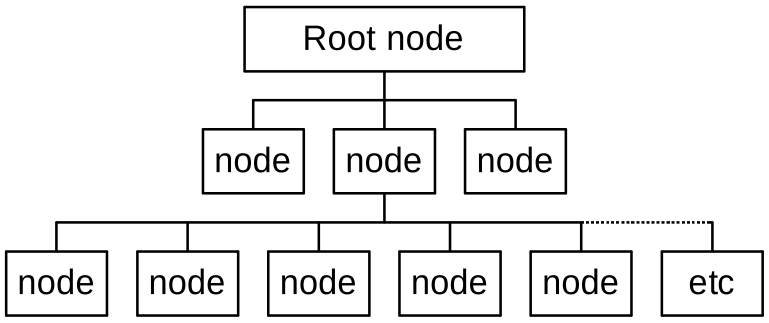


Figure 9.1: DOM tree depicting one parent node which contains the majority of the nodes of the tree as its children.

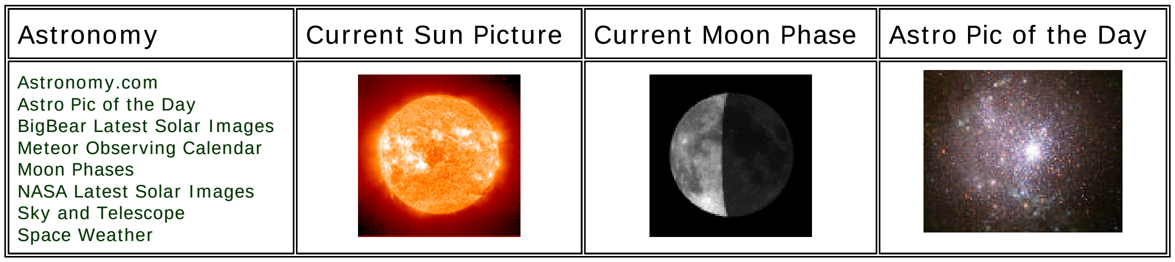
9.1.2 Limitations of the MIU Approach

The MIU approaches we implemented fail to parse certain pages into topical MIUs. One specific example is with the implementation of tables in web pages. Take for example the HTML table in Figure 9.2. The HTML code for this table is shown in Figure 9.3. The table has four header cells: “Astronomy,” “Current Sun Picture,” “Current Moon Phase,” and “Astro Pic of the Day.” Each header cell has one content cell beneath it. If a human were to break this table into MIUs, a logical (to a human) choice would be to group each header cell and its corresponding content cell together as one MIU, resulting is a total of four MIUs. However, if we look at the code for this table we will find that tables in HTML are coded by row. Thus each row is embedded in a table row or <tr> tag. In the structure of our two MIU analysis methods, unless two nodes are children of the same parent node in the HTML DOM tree, there is no way for them to be a part of the same MIU. Since the header cell and content cell will never be children of the same parent node, they will never be a part of the same MIU unless the entire table is a part of the same MIU. This is a major drawback to MIU analysis since many web pages use tables to structure their content.

9.1.3 Limitations of Proximity Scoring

There is one major issue with our implementation of proximity scoring. Hawking et al (see Section 5.2) recommended using a cutoff to eliminate extremely long phrases from contributing to the proximity score. However, this would cause problems with *sticky terms* in a multi-word query. *Sticky terms* are terms in a query that may not have to appear near the other terms although the page is relevant. One example would be “dog door Pasadena.” Presumably, the user’s intention is to look for information on dog doors in Pasadena. Even though Pasadena may not occur near dog door, the page is still relevant. Using cutoffs would eliminate this behavior, unless the term fell within the cutoff range.

In our experiment, we chose to accommodate sticky terms by not using a cutoff. In Section 10.3, we discuss sticky terms in more detail.



Astronomy	Current Sun Picture	Current Moon Phase	Astro Pic of the Day
Astronomy.com Astro Pic of the Day BigBear Latest Solar Images Meteor Observing Calendar Moon Phases NASA Latest Solar Images Sky and Telescope Space Weather			

Figure 9.2: Image of a table coded in HTML. The table has four header cells: “Astronomy,” “Current Sun Picture,” “Current Moon Phase,” and “Astro Pic of the Day.” Each header cell has one content cell beneath it. This table was taken from `www3.hmc.edu/∼ebodine`.

```
<table>
    <tr>
        <td>Astronomy</td>
        <td>Current Sun Picture</td>
        <td>Current Moon Phase</td>
        <td>Astro Pic of the Day</td>
    </tr>
    <tr>
        <td>
            <a href="...">Astronomy.com</a><br />
            <a href="...">Astro Pic of the Day</a><br />
            <a href="...">BigBear Latest Solar Images</a><br />
            <a href="...">Meteor Observing Calendar</a><br />
            <a href="...">Moon Phases</a><br />
            <a href="...">NASA Latest Solar Images</a><br />
            <a href="...">Sky and Telescope</a><br />
            <a href="...">Space Weather</a><br />
        </td>
        <td>
            <a href="..."><img src="..."></a>
        </td>
        <td>
            <a href="..."><img src="..."></a>
        </td>
        <td>
            <a href="..."><img src="..."></a>
        </td>
    </tr>
</table>
```

Figure 9.3: The HTML code for the image in Figure 9.2. This source code was taken from `www3.hmc.edu/∼ebodine`.

Chapter 10

Future Research

In this section we discuss future directions that we think would be fruitful. In the first section, we suggest some additional data analysis that we would have liked to do. In the next section, we present a hybrid MIU analysis procedure that combines bottom up and top down processing. In the final section, we briefly analyze the notion of *sticky queries* and their relationship to proximity scoring.

10.1 Extended Data Analysis

Given more time, there are two more types of data analysis we would perform. First, the MIU re-ranking method we used was different from the MIU re-ranking scheme Li et al used. It would be interesting to implement Li et al's re-ranking method and compare it to the re-ranking method we implemented.

Second, during the analysis phase of our project, we developed a working hypothesis that bottom up MIU analysis is just a very strong version of proximity scoring. Given more time, we would have tested this hypothesis by setting the maximum length of a bottom up MIU and the maximum length of a proximity scoring span to be the same size. If the results returned were significantly similar, it would make a strong case for bottom up MIU analysis being a strong version of proximity scoring.

10.2 Improved MIU Analysis

In our qualitative examination of bottom up and top down MIU analysis, we discovered a number of pages where each method did poorly. One example is pages with tables; an example is discussed in Section 9.1.2. We devised, but did not implement, two extensions to our MIU analysis algorithms designed to improve MIU analysis. Both of these extensions should be fairly easy to implement with the existing code. Unfortunately neither of these methods address the table problem discussed in Section 9.1.2. To address the table problem, we also propose a third, radically different, MIU analysis approach. The new MIU analysis method would work in conjunction with a HTML display rendering engine.

Top Down

```
TopDown($tree, d$)
1  for each $a \in$ children($tree$)
2  do if sim($a, d$) $> \delta$
3        then TopDown($a, d$)
4        else end
5  endfor
```

Dual Threshold Top Down

```
DualTopDown($tree, d$)
1  for each $a \in$ children($tree$)
2  do if sim($a, d$) $> \delta$
3        then DualTopDown($a, d$)
4        else if sim($a, d$) $> \varepsilon$
5                then end
6                else Merge($a, d$)
7  endfor
```

Figure 10.1: The core algorithms for top down MIU analysis and dual threshold top down MIU analysis. This figure highlights the addition of the second threshold value to the top down MIU analysis algorithm. In this algorithm d is the document as a whole.

10.2.1 Dual-Threshold Top Down MIU Analysis

The premise of the dual-threshold top down approach is that there are pages like the example in Section 9.1.1 where the document has a very "flat" DOM tree. On these pages, once top down analysis reaches the root of the linear portion, the page is either very finely segmented into MIUs, or very coarsely (depending on the result of the similarity metric) – there is no intermediate step.

To deal with this case, we propose a *dual-threshold top down approach*. In this approach, there are two threshold values, δ and ε. The δ threshold controls when segments of the tree are too similar, and are split. In contrast, the ε threshold controls when segments of the tree are too dissimilar, and are re-merged.

In this section, we specify the algorithm for the dual-threshold top down algorithm. For brevity, we only present the part of the algorithm that differs from the top down algorithm. For this explanation to make sense, please be familiar with the details of top down MIU analysis from Sections 5.1.3 and 7.1.5. The beginnings of this algorithm are already implemented in the source code for the top down algorithm, although we did not complete the full implementation.

While top down MIU analysis has only two cases, $sim(a, d) > \delta$ and $sim(a, d) \leq \delta$, the dual-threshold MIU analysis method has three cases: $sim(a, d) > \delta$, $\delta \geq sim(a, d) > \varepsilon$, and $sim(a, d) \leq \epsilon$. The behavior of the dual-threshold algorithm in the first case is analogous to the top down algorithm. We split the document if MIU a is too similar to the rest of the document. The second case is also analogous. If the similarity between MIU a and the document is between δ and ε, then we terminate the algorithm, since the MIU is between our similarity thresholds. The last case is novel. If the similarity between MIU a and the document is below ε, then MIU a is too dissimilar to be its own MIU. In this case, we enter a re-merging step. For a concise view of the differences, see Figure 10.1.

The last major detail of the algorithm is the merge step, called on line 6 in Figure 10.1. The merge step looks at the adjacent MIUs and tries to determine which direction to merge. It merges with an adjacent MIU if the following condition holds. The similarity of the

```
MERGE($miu, d$)
 1  $sim_l = \textsc{sim}(\textsc{LeftSibling}(miu), d)$
 2  $sim_r = \textsc{sim}(\textsc{RightSibling}(miu), d)$
 3  if $sim_r < \delta \wedge sim_l < \delta$
 4    then if $\textsc{sim}(\textsc{LeftSibling}(miu), miu) < \textsc{sim}(\textsc{RightSibling}(miu), miu)$
 5           then MERGEADJACENT($miu$, RIGHTSIBLING($miu$))
 6           else MERGEADJACENT($miu$, LEFTSIBLING($miu$))
 7    else if $sim_r < \delta$
 8           then MERGEADJACENT($miu$, RIGHTSIBLING($miu$))
 9    else if $sim_l < \delta$
10           then MERGEADJACENT($miu$, LEFTSIBLING($miu$))
```

Figure 10.2: The merge function for the dual threshold top down MIU analysis algorithm.

adjacent MIU to the document must be less than δ. That is, we do no not merge if the adjacent node will be split by the algorithm. If this condition holds for both adjacent MIUs, then the MIU we are currently examining merges with the MIU it is most similar to, using the *sim* function. If the MIU only has one adjacent node, then it is automatically merged with that node, provided the first condition holds. Figure 10.2 expresses this function in pseudo-code form.

Although untested, we hope that this algorithm would improve top down analysis on pages with flat DOM trees discussed in Section 9.1.1.

10.2.2 Hybrid MIU Analysis

Our proposed hybrid MIU analysis algorithm combines top down and bottom up MIU analysis directly. From our qualitative observations, bottom up MIU analysis performed well on highly linear pages, whereas top down analysis performed poorly (see the preceding section for an example). These are pages where the HTML DOM tree is very "flat." In contrast, top down MIU analysis performed well on highly structured pages. These are pages where much of the layout is based on HTML tables. On these structured pages, bottom up MIU analysis performed poorly.

The intention of our hybrid method is to capitalize on the strengths of each method and offset their weaknesses. To accomplish this, our hybrid method first runs the bottom up algorithm on the web page. The result of the bottom up analysis is then passed as the tag tree to the top down method.

In theory, this algorithm should address the individual weaknesses described above. For example, a highly-linear page will already have been merged into appropriate MIUs when the top down algorithm analyzes it. Likewise, because bottom up MIU analysis cannot merge very many nodes on a highly-structured page, top down analysis can appropriately divide these pages.

10.2.3 Display Rendering MIU Analysis

Our highly speculative *display rendering MIU analysis* algorithm synthesizes the on-screen page layout together with the HTML DOM tree. In this tree, adjacent nodes represent DOM elements that are spatially close. This representation fixes the table problem discussed in 9.1.2. With this representation the table headers, which are spatially close to the content cells, are adjacent, in the tree, to the content cells. Consequently, one might imagine this tree as three-dimensional structure "growing" out of the on-screen display of a page, because spatially close elements will share common parents.

10.3 Sticky Terms

Sticky terms are terms in a query that may not have to appear near the other terms. One example would be "dog door Pasadena." Presumably, the user's intention is to look for information on dog doors in Pasadena. Even though Pasadena may not occur near dog door, the page is still relevant.

Our current implementation of proximity scoring takes sticky terms in queries into account by not setting a limit on the proximity scoring span size (see Section 5.2). Thus, the proximity scoring algorithm may look the entire length of the page to find a span (a string of words containing every term in the query at least once). However, this is not the only approach to dealing with sticky terms in queries. A method could be developed, in which terms identified as sticky need not be included in a span. Then, the span size of the proximity scoring method could have a maximum size, and sticky terms would still be accounted for. Additionally, this would give proximity scoring an advantage over MIU analysis in that it could make use of the knowledge of sticky terms in a query.

Bibliography

[BP] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

> The paper discusses Brin and Page's method for how to make a sufficient search engine given the billions of web pages on the Internet. They claim this was the first paper of its kind that discusses how to handle the web on a large scale. The authors' search engine, Google, debuts in this paper, also. A number of examples of Google's robustness are discussed. The paper does not discuss objectively how good Google is compared to other search engines, but Google's impact presently is clearly felt worldwide.

[CJT] Soumen Chakrabarti, Mukul Joshi, and Vivek Tawde. Enhanced topic distillation using text, markup tags, and hyperlinks. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 208–216. ACM Press, 2001.

> The paper explores a method for improving web search by trying to eliminate needless and irrelevant links in the searching algorithm. This idea of "topic drift" involves creating a tree based on the html structure of the page, and then "cutting off" those branches which the algorithm deems irrelevant. Qualitative testing suggested that the new approach does in fact reduce topic drift, but quantitative testing was to be written in another paper.

[CC] Charles L. A. Clarke and Gordon V. Cormack. Shortest-substring retrieval and ranking. *ACM Transactions on Information Systems (TOIS)*, 18(1):44–78, 2000.

> This paper creates an algebra for Boolean queries and develops an efficient method for ranking documents. The paper has a number of proofs to show that their algebra works, and also describes the large details of the whole model step by step. The paper uses a number of examples to help the reader understand what each defined function does. At the end the paper provides testing examples and asserts that the model is efficient based on evidence, not proof.

[DOM] W3C Document Object Model. `http://www.w3.org/DOM/`, 2003.

[DOMTree] The DOM Tree Structure. `http://www.ifi.uio.no/in-id/2002/student_projects/jorgenn/DOMTree.html`, May 2002.

This web page gives a straightforward description of the structure of a DOM Tree. Each type of node is clearly defined here.

[FLMMT] Leslie Fletcher, Colin Little, Lara Mercurio, Tina Meftah, and Simon Tse. Micro-Information Units: Improving Online Search. Technical report, Harvey Mudd College, 2002. Prepared for Overture Services, Inc.

This is the final report of the Harvey Mudd College summer clinic for Overture Services, Inc. They explored the idea of Micro-Information Units (MIUs) and how they can be used to improve the relevance ordering of web searching. The report includes a summary or their research, their approach, their results and their code.

[NF-HTML] Niklas Frykholm. The HTML Transform library. `http://www.acc.umu.se/~r2d2/programming/perl/html_transform`.

HTML::Transform is a perl module to process and tranform HTML pages. To use the library, the program specifies "handler" routines to call when the library encounters tags in an HTML file.

[Hav] Taher H. Haveliwala. Topic-sensitive PageRank. In *Proceedings of the Eleventh International Conference on World Wide Web*, pages 517–526. ACM Press, 2002.

The original PageRank algorithm returns a single vector that ranks the "importance" of each page. The author modifies the original PageRank idea by developing sixteen different PageRank vectors, each based on a different topic like "sports" or "business." The model is particularly effective when the context of the query is known, for based on that context the model will be able to differentiate between vague terms.

[HT] David Hawking and Paul Thistlewaite. Proximity operators - so near and yet so far. In D.K. Harman, editor, *Proceedings of the Fourth Text REtrieval Conference (TREC-4)*, pages 131–143, Gaithersburg MD, November 1995. U.S. National Institute of Standards and Technology. NIST special publication 500-236.

Hawking and Thistlewaite analyze proximity operators in a text search engine. They define spans of words and show how these are used to compute a simple proximity score. A span is a substring of the document that contains all the words, has a unique starting point, and is as short as possible. The authors also investigate proximity scoring formulas and determine that $\frac{1}{\sqrt{S_i}}$ gives the best precision and recall. S_i is the length of the ith span.

[Jel] Frederick Jelinek. *Statistical Methods for Speech Recognition.* MIT Press, Cambridge, Massachusetts, 1997.

This book discusses the statistical aspects of speech recognition. The basic statistical ideas of speech recognition are covered in length, along with more advanced concepts. In particular, this clinic is interested in Chapter 4: Basic Language Modeling, as the models enable us to create reasonable probabilities for the occurrence of words. The author discusses many heuristics in place of mathematical proofs at times, appealing to the book's practical application toward speech recognition.

[Kee] James P. Keener. The Perron-Frobenius Theorem and the Ranking of Football Teams. *SIAM Review*, 35(1):80–93, 1993.

The author applies the Perron-Frobenius theorem to ranking football teams. Keener discusses four different ways to incorporate the Perron Frobenius theorem. The first approach uses the theorem directly, and creates the football ranking via the Rayleigh method. We adopted this approach and applied it to the creation of our ideal human rankings for each query.

[Kle] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

In this paper, Kleinberg uses the structure of the Internet and link analysis to come up with an effective way of improving relevance ordering in search results. He gives each web page two weights, one conveying the number of web pages that point to it, and another weight which conveys how many web pages it points to.

[LM] R. Lempel and S. Moran. SALSA: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)*, 19(2):131–160, 2001.

This paper compares the results of Kleinberg's HITS algorithm to that of the SALSA. It presents the idea that mutually reinforcing hub and authority scores may not be very meaningful when a tightly knit community (TKC) effect occurs.

[zipfs] Wentian Li. Zipf's Law. `http://linkage.rockefeller.edu/wli/zipf/`, 2002.

This web page describes what Zipf's Law is. It has an extensive list of references to other sources that deal with Zipf's Law.

[LLPH] Xiaoli Li, Bing Liu, Tong-Heng Phang, and Minqing Hu. Web Search based on Micro Information Units. In *Proceedings of the Eleventh International Conference on World Wide Web*, Honolulu, HA, May 2002. World Wide Web Conference.

This is the original poster associated with Li et al's work on MIUs. They define an MIU, but do not present the algorithm used for MIU analysis. See [LPHL] for the full paper.

[LPHL] Xiaoli Li, Tong-Heng Phang, Minqing Hu, and Bing Liu. Using micro information units for Internet search. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, pages 566–573. ACM Press, 2002.

This is the key paper on MIUs and MIU analysis. Li et al present a new approach for analyzing web pages for additional relevance information. They define the concept of an MIU and present an algorithm that performs MIU analysis in a bottom-up fashion. Further, they show that this algorithm, when combined with results from the Google search engine, generates significantly better results in many cases than Google's own results.

[libmba] libmba. `http://www.ioplex.com/~miallen/libmba/`.

The libmba library is a collection of useful C functions and data structures. The library includes a linked list structure, a hash map structure, and a configuration file manager, among other functions.

[MRSWLF] Joel C. Miller, Gregory Rae, Fred Schaefer, Lesley A. Ward, Thomas LoFaro, and Ayman Farahat. Modifications of Kleinberg's HITS algorithm using matrix exponentiation and web log records. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 444–445. ACM Press, 2001.

This paper describes the two modifications to the Kleinberg HITS algorithm. The first modification, Exponentiated Input resolves the problem of the HITS algorithm returning either arbitrary or non-intuitive results. The second modification, Usage Weighted Input, takes into consideration how often users use a certain link in a given time period.

[PBMW] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

The intrinsic importance of a web page may be very subjective, as that may depend on a user's interests and needs. However, one can find the relative importance of web pages by using objective methods that measure a human's attention and interest paid to each web page. The authors present the PageRank method for ranking the relative importance of web pages, and shows how to compute the scores even with a large number of pages. The paper then illustrates the efficiency of the PageRank algorithm by comparing it to an idealized random web surfer.

[HWR] Jonathan Robie Philippe Le Hegaret, Lauren Wood. What is the Document Object Model? `http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/introduction.htm%l`, 2002.

This is a technical web page which is geared towards advanced HTML and XML programmers. It describes what a DOM is, what a DOM is not, where it came from, and other pertinent information. In addition, there are links to other web pages related to topics discussed here.

[RFQWG] Dragomir Radev, Weiguo Fan, Hong Qi, Harris Wu, and Amardeep Grewal. Probabilistic question answering on the web. In *Proceedings of the eleventh international conference on World Wide Web*, pages 408–419. ACM Press, 2002.

The paper explores how a search engine might be able to answer questions, similar to AskJeeves. The model is divided into a number of steps, clearly outlining what needs to be completed. It is a valiant idea, but the execution of it at least in the paper falls short. Intuitively, many types of questions may be very difficult to answer just by searching through documents and identifying key words.

[TidyLib] HTML Tidy Library Project. `http://tidy.sourceforge.net/`.

The HTML Tidy library project publishes the TidyLib HTML processing library. This library produces data structures for the HTML DOM Tree.

Appendix A

Related Research

A.1 Kleinberg's HITS Algorithm

The Kleinberg Algorithm [Kle] is a local Internet search algorithm that uses the text-based searches and the hyperlinked environment of the Internet in order to return relevant search results to a given query. The Kleinberg algorithm, also known as the Hyperlink Induced Topic Search (HITS) algorithm is built on the idea that the purpose of a website is either to provide information on a certain topic, or to provide links to other pages that have information on that same topic. So under this algorithm, given a query, each web page is given two different weights, the *authority* weight and the *hub* weight. The authority weight measures how good the information on the given subject is. The hub weight measures how good of a hub the web page is. That is, for a given query, it measures how good the information is on the pages the page being weighted points to. This algorithm is a mutually reinforcing relationship, if the web page provides links to good authorities, then it would be given a higher hub weight, or if the web page provides links to sites with lower authority weight, then it would be given a lower hub weight.

When a query is typed into a search engine, an ideal set of search results would (1) be relatively small, (2) be rich in relevant pages, and (3) contain most or many of the strongest authorities on the query topic.

In order to accomplish this, Kleinberg first uses a text-based search engine such as Alta-Vista, and takes the top t pages (where t is usually set to about 200) returned for some query. This set of web pages, called the *root set*, satisfies both (1) and (2). However, the root set does not necessarily contain all of the highly weighted authorities. For example, if the query in a text-based search was "red T-shirt," the top search results returned are web pages containing the term "red T-shirt" most frequently. A store specializing in making red T-shirts should presumably be in the top t returned results. However, since these sites do not contain the word "red T-shirt" as frequently as many other irrelevant sites, this may not necessarily be the case. In order to circumvent this problem, the root set is expanded to include more potentially strong authorities on a particular query. This is done by including all the web pages that the root set points to, and also by including at most d web pages that point to the root set. In his paper, Kleinberg set d equal to fifty. This expanded set, called the *base set*, satisfies (1), (2), and (3).

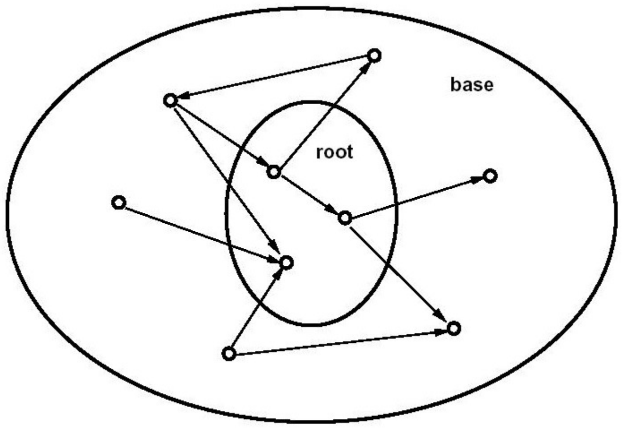


Figure A.1: This figure, taken from [Kle], illustrates the process in which the root set expands to the base set. All of the web pages which the root set points to as well as d web pages pointing into the root set is included in this expanded set.

The Kleinberg Algorithm then calculates the authority and hub weights for each web page in the base set. The hub weight at time $k+1$ of web page p is defined as the sum of the authority weights at time k of the pages to which p points. The authority weight at time $k+1$ of web page q is defined as the sum of the hub weights at time k of all the pages that point to q. This is an iterative process, where the Kleinberg Algorithm begins with hub and authority weights each being normalized, so that the sum of the squares of the authority and hub weights are each equal to one. Then, after each iteration the hub and authority scores are normalized again so that the sum of their squares is equal to one. Kleinberg also showed that this algorithm was guaranteed to converge. Often, about twenty iterations are sufficient to stabilize the authority and hub scores. Then the authority scores are used as the page rankings.

A.2 Other Web Search Algorithms

We examined the Exponentiated Kleinberg algorithm devised by the HNC Clinic Team [MRSWLF], and the SALSA algorithm [LM].

A.2.1 Exponentiated Kleinberg

Hecht-Nielsen Corporation (HNC) Software sponsored an HMC mathematics clinic in the 1999-2000 school year [MRSWLF]. HNC specializes in software used in credit card fraud detection. The division of HNC Software that the team worked for was eHNC, which dealt with providing e-commerce solutions. This division wanted to make e-commerce websites more efficient, which would in turn boost sales potential.

The HMC clinic team started off by examining techniques in link analysis, and applying them to usage data provided by eHNC. Usage data is data that gives quantitative results about the usage of a website. The one that was used in this clinic was a web server log, in which the data gave information such as who was requesting which page, and from which page the user came. Initially the HNC clinic investigated ways to implement this as a tool in fraud detection, focusing particularly on cellular phone fraud detection. As the year progressed, the goal changed from the above to applying methods in link analysis to find subnetworks of related individuals, and using social network theory to predict the existence of links between individuals in a network.

The Kleinberg Algorithm was implemented with the aim of identifying cliques or subnetworks of related individuals. The clinic team also implemented some new link analysis techniques. The two modifications they made to the Kleinberg Algorithm were *Exponentiated Input* to the Kleinberg Algorithm, and the *Usage Weighted* Kleinberg Algorithm. The team proved that Exponentiated Input resolves two problems suffered by the Kleinberg Algorithm, which were that the final hub and authority weights are not always independent of the initial seed, and that zero weights can be assigned inappropriately. The HNC clinic also developed ways to analyze the usage patterns of a website by using compression ratios.

A.2.2 SALSA and TKC Effect

The *Tightly-Knit Community (TKC) Effect* occurs when a densely interlinked cluster of web pages receive high authority scores although these web pages do not have good information on the particular query. This is not always bad. However, it can cause other more highly relevant web pages to have lower scores relative to those web pages that have high scores due to the skew created by the TKC Effect. For example, if there were a community x with a small number of hubs and authorities, where each of the hubs points to all the authorities, then each of the authorities would be given a high authority score. However, if there were another, larger community y with web pages that are more relevant to the subject, but at the same time, each hub points to only some of the authorities in y, the authorities in y would get a lower score because y is not as densely interconnected as x is.

The Stochastic Approach for Link-Structure Analysis (SALSA) work of Lempel and Moran [LM] is an algorithm that implements Markov chains, and stochastic properties of random walks performed on the root set. If we have a graph where distinct WWW links from the root set are the nodes, then a random walk on this graph will, with a high probability, visit the relevant web page authorities. The state transitions in these graphs are the result of traversing two hyperlinks in a row. By analyzing these two chains, a hub and authority score could be given. The nodes that are being visited most frequently will determine the top authorities and hubs. Each node corresponds to its own Markov chain, which is an irreducible, aperiodic stochastic matrix. By the Ergodic Theorem, the principal eigenvector of an irreducible, aperiodic stochastic matrix is the stationary distribution of the underlying Markov chain. Its highest entries correspond to the sites that are most frequently visited by the random walk, and thus correspond to the strongest authorities and hubs.

A.3 Summer Clinic Summary

Our summary of the 2002 Overture Summer Clinic is divided into three sections. First, we present the motivation and the work of the Summer Clinic. Then, we present a summary of the code behind their implementation. Finally, we conclude with an analysis of their work and results.

A.3.1 Ideas

The summer clinic team decided to use Micro Information Units (MIUs) in order to improve web searches. MIUs are blocks of text on a web page that contain only one topic. This was a new concept introduced in 2001 by Li et al [LPHL]. The summer clinic implemented the MIU idea somewhat differently from Li et al. That is, instead of creating MIUs during query time, the clinic team proposed pre-processing the entire web into MIUs, and then running a search on these. Compare this idea to a search without using MIUs, where one might find the two words of a query on a web page but in different paragraphs, which have nothing to do with each other. Presumably, if a search is done on blocks that contain only one topic, then the search should be better able to determine if the text is relevant.

The summer clinic team first converted a test subset of the web into MIUs. This set consisted of about 2.5 million web pages from the Stanford WebBase. The team took each web page and ran it through their version of the MIU analysis, which was to break pages up according to the existing paragraph breaks (<p>). Subsequently, they compared adjacent MIUs, and merged these two MIUs if the texts shared similar terms. To decide which adjacent paragraphs would merge, the team needed a way to measure similarity between texts: that is, to decide whether adjacent paragraphs were about the same topic. For this, the team used the VSM method, which will be explained below.

In order to convert a test set of the web into MIUs, the team needed some way to measure similarity between texts. For this the team used the *vector space model*, or *VSM*. VSM turns texts into vectors, and then uses some function that compares how similar they are. The team used the *cosine similarity function* in order to do this. They took the inner product of the two texts (in vector form), appropriately weighted to adjust for the fact that smaller web pages tend to be overlooked and larger pages appear more frequently. Through this process, a normalized score was generated, signifying “how similar” the two blocks of texts were. If the score exceeded some threshold defined by the team, then the two blocks were merged. The team chose to compare only adjacent texts with each other rather than comparing all texts pairwise to each other. This cut down on run time and intuitively makes more sense since texts far away from each other are less likely to be relevant compared to texts that are adjacent. Once the program stopped merging texts, the algorithm stopped. This took about 20-30 iterations at most.

The summer clinic's MIU breakup, or MIU analysis, should improve the root sets and the authority scores. In theory, breaking up the web pages into MIUs reduces the chances of a search engine returning results of a certain query that are on the wrong topic. This is true because the MIU analysis weeds out the web pages that contain the query terms in different MIUs. Hence, the improvement in the root set. Since we have a more precise, relevant root set, this also results in the improvement of the assignment of authority scores.

A.3.2 Our Review of the Summer Code

The Summer 2002 Overture Services Clinic report [FLMMT] included the full source code for their implementation of the Kleinberg and MIU-seeded Kleinberg algorithms, along with supplementary documentation. Although several of their command line maintenance tools are not included in their report, our team has access to the computers they used over the summer, which still include these tools and all of their data. The code performs the basic Kleinberg and MIU-seeded Kleinberg search. In addition, it exploits parallelism in various processing stages.

Unfortunately, the structure of the code makes it both hard to read and hard to modify. This limitation is understandable in light of the time restrictions of the Summer Clinic. The team divided the code into multiple files where each file acts as a subroutine, taking some input and generating some output. There is no reuse between files, and the code uses the copy-and-paste action a lot. Consequently, any change needs to be applied to every repeated occurrence, rather than to only one location. Also, there are many magic numbers and magic strings throughout the code. These are values used simply as constants, sometimes multiple times. For any changes, one must change all of the specific values, instead of only changing a common variable. One notable occurrence of this is with file names. Instead of a shared list of file names, each program contains all of the file names as inline strings. As a result, the code lacks modularity. We would need to alter large portions of the existing code even for relatively simple alterations.

The language the Summer Clinic chose to implement their algorithm was Perl. Perl is a difficult language for future programmers to support. There are many different ways to accomplish the same task, and so extremely good documentation is essential with Perl code. The Summer Clinic team did not fully document their code. Throughout the code, the comments assume a reader who is familiar with the Perl features used. These comments, however, do not always explain how these features are used in the code. Also, there is no documentation of the intermediate files created by the various processing steps.

A.3.2.1 Decision to Write our Own Code

Originally, we thought we could modify the Summer code to abstract common functionality and create generalized input-output routines. These routines would correspond to the sections in our pipeline diagrams (see Section 4.1).

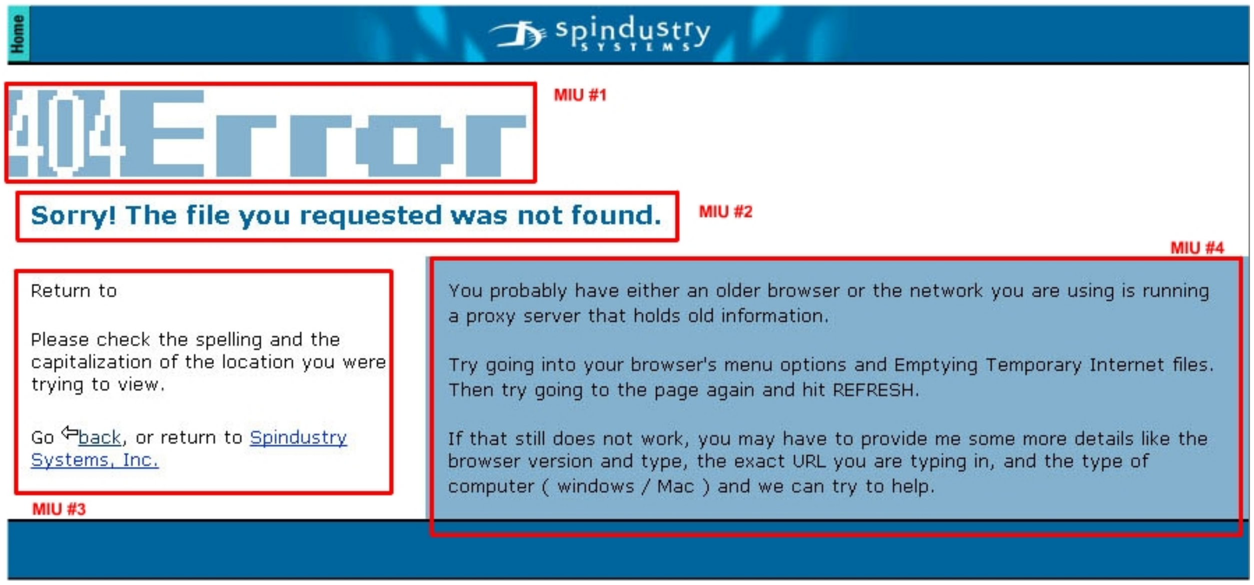
Further review showed this not to be feasible. The existing code was too interdependent to easily modify. However, this is not a significant problem; the course of our project changed (see Chapter 2) and it no longer requires an implementation of the Kleinberg algorithm.

Although the existing code contained a working version of MIU analysis, this analysis depended on a global word list for all the pages in the data set. Thus, even abstracting simply the MIU analysis would have been difficult and time consuming. Consequently, we have decided to create entirely new code for the project, although we may base ideas and implementations on the Summer Clinic's code.

A.3.3 Our Report On The Summer Code

The Summer Clinic broke up the web pages pre-query. They broke up these pages into MIUs using the html tags, splitting the page up by paragraph breaks (<p>). Then, they merged the adjacent MIUs if they contained enough of the same terms a certain number of times. We looked in detail at the code's effect on 4 web pages, and found that in some cases, the MIUs did not merge correctly. That is, even though some of the neighboring MIUs dealt with the same topic, they did not merge. Also, there were instances when the initial MIU analysis did not break up the web page into enough MIUs. Hence, the MIU analysis failed to separate the text into independent topics. In addition, extraneous HTML code such as comments, were sometimes included in the MIU analysis when they should not have been.

One of the web pages we looked at from the Summer Clinic's database was `http://www.ahanews.com/robots.txt`. After some analysis of the code, and examination of a web page that was run through the Summer Clinic's MIU analysis, we made a few observations. As we stated above, the MIUs were split up according to paragraph breaks. In this case, the code's break up agreed with the break up we would have reached by hand. However, in general, this will not always be the case. Also, the Summer Clinic's code included extraneous html code which included comments, words in the html tags (e.g. topmargin, leftmargin, and bgcolor). Figure A.2 shows this web page after going through the MIU analysis.



Sorry! The file you requested was not found.

Return to

Please check the spelling and the capitalization of the location you were trying to view.

Go back, or return to Spindustry Systems, Inc.

You probably have either an older browser or the network you are using is running a proxy server that holds old information.

Try going into your browser's menu options and Emptying Temporary Internet files. Then try going to the page again and hit REFRESH.

If that still does not work, you may have to provide me some more details like the browser version and type, the exact URL you are typing in, and the type of computer (windows / Mac) and we can try to help.

Figure A.2: This image is the web page, http://www.ahanews.com/robots.txt, which is broken up into MIUs according to the Summer Clinic's version of MIU analysis.

Our team also looked at how many pages were split into n MIUs. We took the Summer Clinic's data set that they had run through their MIU analysis. Then we took these web pages that were broken up into MIUs, and ran these through our code, which tells us the number of MIUs in each web page. From this, we found that that over half of these web pages had only one MIU. On the extreme side, we found one web page with 400 pages that was broken up into 1381 MIUs. Figure A.3 is a histogram of our findings.

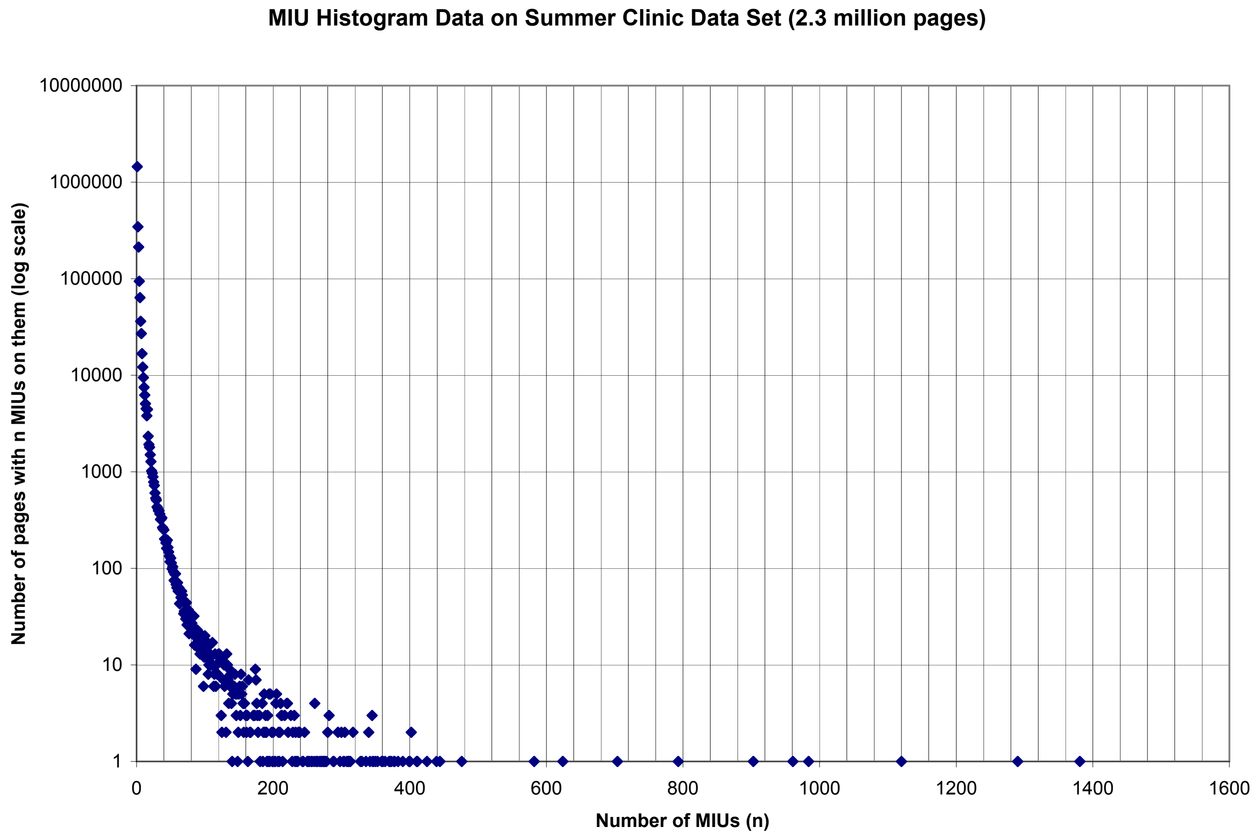


Figure A.3: This is a histogram of log of the number of pages with n MIUs vs. number of MIUs (n)

We conclude that the links in a given web page would end up in whatever MIU they just happen to fall under according to the Summer Clinic's method. Thus, the links do not always match up with the particular MIUs they should be in. This, in turn, would result in inaccurate search results, because Kleinberg's Algorithm is dependent upon the links on a web page. Since each MIU is treated as its own web page, the Kleinberg Algorithm would end up returning inaccurate results.

A.4 Topic Sensitive Search

A user's intent when entering a query may be ambiguous, depending on the words entered. For example the query "blues" does not specify if the user intended to search for music or medical depression. Ambiguous queries may be another issue for the Overture Clinic team, as the overall goal of the project is to improve the relevance ordering of web searching. Haveliwala [Hav] presents an idea that deals with this issue, and may lead to future research and work for the team.

The original PageRank algorithm computes the relative importance of each web page and returns the rankings in a single PageRank vector before any particular search query is done. Haveliwala [Hav] proposes creating a *set* of PageRank vectors to divide the web up by topics, and doing a search on these topics in order to make the desired results more specific. In certain instances this approach is very useful because the context of the query can eliminate ambiguity. For example, if a user performs a query by highlighting the term "architecture" from a web page related to creative building designs, then it is reasonable to want the results to be related to buildings as opposed to the "architecture" of CPU design.

The subject matter of the web page can be used to find more of the right links. The authors also mention other contexts, such as the user's declared interests, or the user's past history of web searches. The paper goes into some detail about the algorithm used, and uses actual volunteers to argue that their results are more specific than an unbiased PageRank search. This concept might be an interesting way to improve upon or modify the work previously done, as it does not appear to be unique to the PageRank algorithm itself.