

Incremental Web Search: Tracking Changes in the Web

by

Ziyang Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
May 2006

Ernest Davis

© Ziyang Wang

All Rights Reserved, 2006

*Dedicated to my dear parents Shidong Wang, Weiqing Gong
and my lovely wife Xiang Jin who blessed and supported me*

Acknowledgments

Many thanks to my advisor Professor Ernest Davis for all of his kind help with my work. He is very nice and patient in helping his students. I am very lucky to find him as my advisor. Thanks to other committee members of my thesis: Prof. Panagiotis Ipeirotis, Prof. Zvi Kedem, Prof. Dennis Shasha and Prof. Ralph Grishman. They gave many valuable advices in reading the draft of the thesis. Some of this thesis work are presented at different conferences and workshops. Many anonymous reviewers gave valuable suggestions to improve my work. During my Ph.D. studies, I did internships at Avaya Research Labs, IBM Watson Research Center and Amazon Inc. The team members in these companies also helped me polish the ideas related to my work. I would like to thank my wife, Xiang Jin, for her years support during my Ph.D. study. She encouraged and helped me a lot to concentrate on my study and made my life go smooth for years. Now she is going to get a degree in Carnegie Mellon University too. I would like to congratulate her great achievements. Finally, special thanks to the staffs, Anina Karmen and Rosemary Amico, in the department of computer science of New York University. They are very helpful in dealing with my academic issues during the last five years.

Abstract

A large amount of new information is posted on the Web every day. Large-scale web search engines often update their index slowly and are unable to present such information in a timely manner. In this thesis, we present our solutions of searching new information from the web by tracking the changes of web documents.

First, we present the algorithms and techniques useful for solving the following problems: detecting web pages that have changed, extracting changes from different versions of a web page, and evaluating the significance of web changes. We propose a two-level change detector: MetaDetector and Content-Detector. The combined detector successfully reduces network traffic by 67%. Our algorithm for extracting web changes consists of three steps: document tree construction, document tree encoding and tree matching. It has linear time complexity and extracts effectively the changed content from different versions of a web page. In order to evaluate web changes, we propose a unified ranking framework combining three metrics: popularity ranking, quality ranking and evolution ranking. Our methods can identify and deliver important new information in a timely manner.

Second, we present an application using the techniques and algorithms we developed, named “Web Daily News Assistant (WebDNA): finding what’s new

on Your Web.” It is a search tool that helps community users search new information on their community web. Currently WebDNA is deployed on the New York University web site.

Third, we model the changes of web documents using survival analysis. Modeling web changes is useful for web crawler scheduling and web caching. Currently people model changes to web pages as a Poisson Process, and use a necessarily incomplete detection history to estimate the true frequencies of changes. However, other features that can be used to predict change frequency have not previously been studied. Our analysis shows that PageRank value is a good predictor. Statistically, the change frequency is a function proportional to $\exp[0.36 \cdot (\ln(\textit{PageRank}) + C)]$. We further study the problem of combining the predictor and change history into a unified framework. An improved estimator of change frequency is presented, which successfully reduces the error by 27.3% when the change history is short.

Contents

Dedication	iv
Acknowledgments	v
Abstract	vi
List of Figures	xii
List of Tables	xiv
List of Appendices	xv
1 Introduction	1
1.1 Motivation	1
1.2 Support from the statistical data of web evolution studies	3
1.3 Our Contributions	5
2 Techniques and algorithms	8
2.1 Problems of searching for new information over the web	8
2.1.1 The definition of indexable change of web pages	8
2.1.2 The framework and problems of incremental web search .	9
2.2 Web document change detection	13

2.2.1	Level 1: <i>MetaDetector</i>	13
2.2.2	Level 2: <i>ContentDetector</i>	16
2.2.3	The effectiveness of change detector	18
2.3	Extracting web changes between different versions	19
2.3.1	Document tree construction	21
2.3.2	A bottom-up tree encoding for hypertext documents: <i>HT-encoding</i>	27
2.3.3	An efficient algorithm for tree matching	29
2.3.4	Complexity of algorithms	32
2.3.5	Information segmentation	35
2.3.6	Demo	35
2.4	A framework of evaluating web changes	39
2.4.1	Popularity ranking	40
2.4.2	Content-based ranking	45
2.4.3	Structured evolution ranking	46
2.4.4	Unified ranking	48
2.5	Summary	49
3	Application: Web Daily News Assistant	50
3.1	Design	50
3.2	Architecture	53
3.3	Building the initial data	56
3.4	An event-driven incremental web crawler for WebDNA	59
3.4.1	Determining the crawling rate	62
3.5	Retrieving web changes from database	66
3.5.1	Full-text keyword search	67

3.6	Developing tools	68
4	Modeling Web Changes and Analysis	70
4.1	Introduction	70
4.2	Background	72
4.2.1	Change frequency and change history	72
4.3	Modeling web changes	72
4.3.1	Review of frequency estimator based on detecting history	73
4.3.2	The disadvantage of detection based frequency estimator	74
4.3.3	Predicting the change frequency of web pages	75
4.3.4	An improved change frequency estimator	78
4.4	Experiments	79
4.4.1	Simple regressions on uncensored data	80
4.4.2	Cox regression on both uncensored and censored data . .	81
4.4.3	The effectiveness of improved estimator	83
4.5	Conclusion	85
5	Related work	87
5.1	Searching the Web	87
5.2	Web evolution studies	90
5.3	Web crawling and synchronization	92
5.4	Hypertext information extraction	94
5.5	Web ranking	96
6	Conclusion	98
	Appendices	101

List of Figures

2.1	An overview of incremental search.	10
2.2	The control flow of <i>MetaDetector</i>	15
2.3	Detection order of meta data.	17
2.4	An example of modification areas on web pages.	20
2.5	Misalignments of HTML tags.	22
2.6	Tag structure transformation for <i>OptTags</i> and <i>ForbTags</i>	23
2.7	Two versions of the home page of New York Times.	37
2.8	The changed content extracted.	38
2.9	Information segments generated.	39
2.10	The framework of ranking web changes.	40
2.11	The selection of α	48
3.1	The user interface of WebDNA.	52
3.2	The architecture of WebDNA.	54
3.3	HTTP response and media type distribution of URLs in NYU domain.	58
3.4	Top sub-domains in NYU web.	59
3.5	The eventloop of incremental crawler.	61

3.6	The loading ratio and crawling load on web servers. (The error bar is mean +/- standard deviation).	63
4.1	The ratio of standard error of change frequency estimator. . . .	75
4.2	The distribution of URLs.	81
4.3	Simple exponential regression.	82
4.4	Cox regression on different history data sets.	84
4.5	The error ratios of change history based estimator.	85
4.6	The error ratios of the improved estimator.	86
5.1	General search engine architecture (from [9])	88
C.1	Top 100 most frequent words in web changes of NYU Web. . . .	107
C.2	Top 100 most frequent words in the NYU Website.	108

List of Tables

2.1	Presence of meta data	15
2.2	Percentage of changed documents found by different detection methods.	19
2.3	HTML tags for separating content	36

List of Appendices

Appendix A	101
Estimating the number of distinct trees	
Appendix B	103
Computing minimum backward distance	
Appendix C	106
Top frequent words in the web pages and web changes of the NYU web site	
Appendix D	109
Estimating the lifetime of web changes on a single web page	
Appendix E	111
The probabilities for sampling web pages	

Chapter 1

Introduction

1.1 Motivation

The World Wide Web is an enormous information source useful to millions. Many users use a search engine to find the information they need when surfing the web. According to the Pew Internet & American Life Project [35], there are over 107 million users of Web search engines in United States alone, and they made over 3.9 billion queries in the month of June 2004. A search engine usually takes a keyword query from user and returns a ranked list of the most relevant web documents. There is no question that the Web is huge and hard to deal with. In order to provide a high coverage for searching, search engine maintains a huge index of web pages and updates it regularly. For example, Google claims that it had indexed over 8 billion pages as of June 2005. Although processor speeds are increasing and hardware is getting less expensive every day, updating such a huge index in a short time is a very challenging task. Typically, it takes from weeks to a couple of months for a comprehensive update. Although the precise information of how often the search engines update the

index is their commercial secret, the huge size prohibits a short term update. For information that is relatively static on the Web, e.g. “Java API documentation”, and navigational queries of long-lived URLs, e.g. “CNN”, this delay does not matter much. However, the Web changes rapidly. According to the experiments performed by Fretterly et al. [36] and Ntoulas et al. [54], 15% – 25% of web pages change at least once each week. Searching information on frequently updated pages will fail using an obsolete index.

In this thesis we consider a different way of searching the Web. Rather than searching the Web as an entire information library, searching can be targeted toward new information appearing on the Web. During the evolution of the Web, a large amount of new information emerges on the Web every day. We need a high access rate to download such information and present it to users in a timely manner. The traditional download-and-index approach is not an efficient solution in dealing with such information.

Searching for new information from media is not a new area of study. News search engines, such as Google News [2], provide search tools for public news media. They retrieve news articles from the databases of various news media and download articles from many news web sites, build an index and present a search service. But this technique applies only to data sources presented by public news media which are created by professionals, published at standard locations and are of general public interests. What has not been well studied is how to retrieve and organize the innumerable updates that are constantly posted on the web. New information of the Web can be classified into two categories: changes to existing web pages and newly created pages. The underlying technologies to process information of these two categories might be very different. We focus our studies on the first category only while leaving the second one open. In this

thesis, we present our approach of how to search for new information from the Web by retrieving and presenting the changes in web documents. The techniques of searching new information from web changes can be very useful in many applications. We give two examples here: finding new information in a local web site; and incremental indexing of web data. Localized web or community web are closely related to people's daily life. Much new information on those webs, such as event announcements, has very short lifetime. Retrieving changes of web documents at high frequency can quickly bring such information to users. Incremental indexing of web data can maintain a fresh web index continuously. Indexing by snapshot requires huge amount of resources. By comparing different versions of web documents, we can reduce the cost and make indexing process much faster.

1.2 Support from the statistical data of web evolution studies

Why is retrieving and managing web changes an effective method for retrieving new information from the web? Evolution studies of the Web [36, 54, 22, 13] demonstrate the following: *Although the Web is growing and changing fast, the absolute amount of changed content on existing web pages during a short period is significantly smaller than the total amount of content in the Web.* We review the results of two recent studies.

- Ntoulas et al. [54] collected a historical database for the web by downloading 154 popular Web sites (e.g., acm.org, hp.com and oreilly.com) every week from October 2002 until October 2003, for a total of 51 weeks. The

average number of web pages downloaded weekly was 4.4 million. The experiments show that a significant fraction (around 50%) of web pages remain completely unchanged during the entire period they studied. To measure the degree of change, they compute the shingles¹ of each documents and measure the difference of shingles between different versions of web documents [15, 18]. They show that many of the pages that do change, undergo only minor changes in their content: even after a whole year, 50% of the changed pages are less than 5% different from their initial version.

- Fretterly et al. [36] performed a large crawl that downloaded 151 million HTML pages. They then attempted to fetch each of these 151 million HTML pages ten more times over a span of ten weeks during Dec. 2002 to Mar. 2003. For each version of each document, they compute the checksum and shingles to measure the degree of change. The degree of change is categorized into 6 groups: complete change (no common singles), large change (less than 30% common shingles), medium change (30%-70% common shingles), small change (70%-99% common shingles), no text change (100% common shingles), and no change (same checksum). Experiments show that about 76% of all pages fall into the groups of no text change and no change. The percentage for the group of small change is around 16% while the percentace for groups of complete change and

¹A shingle is a k -word subsequence, for some $k > 1$. The shingle method uses a window of size k moving from the beginning of a document to the end and records all shingles into a vector. For web documents, mark-ups are removed before the shingle method is used. Quantitatively, the similarity of two documents is defined to be the number of distinct shingles appearing in both documents divided by the total number of distinct shingles.

large change is only 3%.

The above results are very supportive to our studies. They suggest that incremental method may be very effective in updating web indexes, and that searching for new information appearing on the web by retrieving the changes will require a small amount of data processing as compared to the huge size of the Web.

1.3 Our Contributions

In Chapter 2, we studied a set of problems emerging in the application development of searching new information from the web, and for each problem, we present a practical solution and analyze its efficiency. The problems and solutions are:

- How can changes in web pages be detected effectively? We detect changes on-line using a two-level detecting method. Level 1 detection uses the meta information of web documents retrieved from web servers to determine whether a document is changed. If Level 1 detection fails, we use Level 2 detection, which detect changes using a hypertext document tree encoding of the document. The overall quality of our detection method is superior to the naive change detection which downloads all pages and detect changes offline. It reduces the network cost by about 67%. Our detection method can also distinguish indexable and non-indexable changes. Among all candidate pages that are modified, about 51% are found having indexable content changes.
- How can changes between different versions of web documents be extracted

effectively? We present an efficient and effective algorithm for comparing different versions of web documents. The change extractor based on this algorithm includes three phases: document tree construction, bottom-up tree encoding and top-down matching. Our extractor is very effective for real web data in practice and has linear scalability.

- How should new information be evaluated? We rank new information by combining the results of three ranking schemes: popularity ranking, quality ranking and evolution ranking. Popularity ranking evaluates the global importance of web pages that present the changes in the Web. An improved link-based ranking algorithm is presented for measuring the popularity of web documents. Quality ranking evaluates the content information of web changes in favor of how much information they carry and how timely the new information is. Evolution ranking evaluates new information based on the time-stamp of changes and show how the importance of web changes decreases over time.

In Chapter 3, we present an application using the techniques and algorithms we developed, named “Web Daily News Assistant (WebDNA): finding what’s new on Your Web”. WebDNA is a web search tool that helps community users search new information on their community web. We deployed this application on the Web site of New York University.

In Chapter 4, we discuss how to allocate and schedule crawling resources. We model the changes of web documents using survival analysis. We find that the popularity metrics of web pages is a good predictor of web changes. We show that such a predictor can be utilized to improve the change frequency estimator by 20% when the detection history is incomplete. Furthermore, we show how to

improve the freshness of the web index using the improved frequency estimator and quality metrics of web documents.

Chapter 2

Techniques and algorithms

2.1 Problems of searching for new information over the web

2.1.1 The definition of indexable change of web pages

First of all, what is the definition of a change on a web page?

Definition 1. *If a web page P is modified at time t , the change of P at time t is defined as the new text data added on the version at t compared with the latest version before t .*

According to this definition, a change of a web page is made up of two kinds of information: the event of modification at time t and the new content added in the new version. Since the content removed during a change can be considered as obsolete information, we ignore it in our study. From the perspective of a search engine, such a definition might not be very useful. First, a web page can be inaccessible to search engines (e.g. the page is published on a local network

only, or is generated by a query from a database); second, a modification of a web page might not introduce any new information that can be indexed by a search engine. Therefore, we use a stronger notation to describe a change of web page: *indexable change*. The term “indexable” has two levels of meanings: a web page is considered as indexable if the page is accessible from the internet through a unique URL; a piece of HTML content is considered as indexable if the text data within the content contains non-stop words. Therefore, *indexable change* is defined as following:

Definition 2. *An indexable change is a change to textual content on an accessible web page.*

In the rest of this thesis, unless stated otherwise, when we talk about a *web change*, we always refer to an *indexable change* of a web document.

2.1.2 The framework and problems of incremental web search

In general, our approach to incremental search uses a framework similar to a general search engine (Figure 5.1, Section 5.1). However, new problems and underlying techniques arise within each of the major components. Figure 2.1 is an overview of searching new information from a data processing view. We divide the retrieval process into three phases and identify the key problems within each phase.

- **Phase 1: fetching modified documents from the Internet**

Our first task is to find and download the candidate pages that may contain indexable changes during crawling. The candidate pages are those

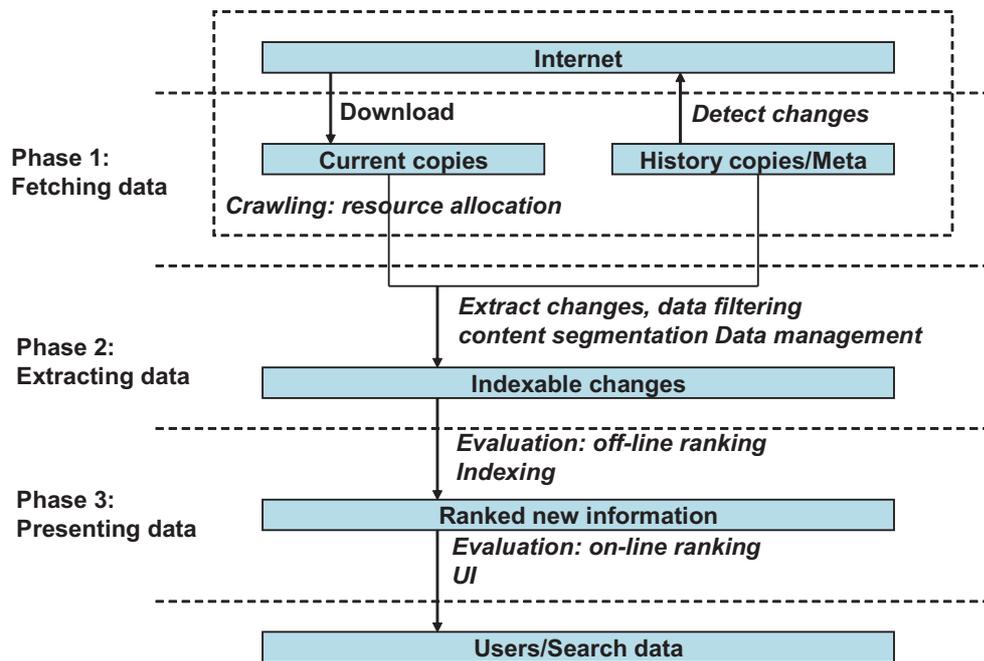


Figure 2.1: An overview of incremental search.

that have been modified since the last visit. Therefore the first problem is how to effectively detect the modification of web pages. The goal is to find the candidate pages that have been modified since the last visit accurately while reducing the average cost as much as possible. We only learn that a web page has changed when we detect that a new version differs from a previous version. When such a change is detected, we do not, of course, know when the change was made, or how many times the page was changed. Therefore, searching for new information requires high detecting frequency in order to reduce the delay of two successive detec-

tions and increase the freshness of local data. How to reduce the average cost of single detection is a big challenge.

- **Phase 2: extracting and processing web changes**

The second problem is how to effectively extract indexable changes by comparing the current version with the last version of the document. There are two challenges in designing algorithm for this problem. First, unlike pure text data, web documents contain hypertext markups as well as text data. Most indexable content is provided in the text data. However, the markup data provides a structural view of the text data. The string algorithms for comparing pure text documents, e.g. [50], are inappropriate for comparing web documents. Second, the volume of web data is huge. Algorithms dealing with web data must be very efficient. Complex algorithms for processing structural data are not practical. Subsequent data processing, such as data filtering and content segmentation, can be applied after the changes have been extracted.

- **Phase 3: presenting new information**

A search engine usually presents results to a query through a ranked list. The quality of presentation relies on the quality of the evaluation of data. In presenting web changes, rather than ranking web documents, we consider the problem of ranking web changes. We know that the information carried by web changes differs from that carried by web documents. Some meta information differs too, such as the location of the change and the modification timestamp. Due to these differences, there are several sub-problems as described in the following:

- How to rank changes between different web documents.
- How to rank changes appearing at different locations on a single web page.
- How to rank changes appearing at different time on a single web page.

In presenting results to users, it is not proper using multiple ranked lists of different categories. We need to combine multiple evaluation strategies to provide a unified evaluation. We need solutions for each of the sub-problems and study how these solutions can be combined to an comprehensive evaluation.

In the following of this chapter, we will present the solutions to these problems and discuss their effectiveness. The data we use for our analysis is collected from the web of New York University by the application presented in Chapter 3. It includes a snapshot of about 240,000 web pages, the link structure of these pages, and the complete change history of 60,000 web pages from March 2005 to November 2005. Although the NYU web site is much smaller than the global web, it is structurally similar¹. The ratio between the number of hyperlinks and URLs is 7.4, which is in agreement with previous work [17]. The study of the in-link and out-link distributions shows they obeys power law with power coefficients of 1.94 and 2.24. Both of them agree well with previous work studied by Broder et al. [17] and Barabasi and Albert [11]. The average size of the web documents is 10.5K bytes similar to the results given by Fretterly et al. [36].

¹Kumar et al. [45] showed that the structural similarity between local webs and the global web is typical.

2.2 Web document change detection

We use two detectors to find candidate web pages that has been modified since last visit: *MetaDetector* and *ContentDetector*. *MetaDetector* performs on-line detection using meta data given by HTTP HEAD responses. If it fails to determine whether a web page has been changed, we download the web page and start *ContentDetector* that performs detection using the content of the page.

2.2.1 Level 1: *MetaDetector*

There are two different types of HTTP messages: a Request Message sent by client, and a Response Message returned by server. Each HTTP message can include a list of header fields. In HTTP/1.1 (RFC 2616) [7], header fields are divided into four categories: *General Header Fields*, which apply to both request and response messages, but do not apply to the entity transferred; *Request Header Fields*, which allow the client to pass additional information about the request, and about the client itself, to the server; *Response Header Fields*, which allow the server to pass additional information about the response which cannot be placed in the Status-Line; *Entity Header Fields*, which usually present the meta information of the entity body. Among these headers, entity headers and cache control headers can be used to determine whether the content has been modified and whether we should update the local cache.

Related entity headers are:

- *Last-Modified* – the latest timestamp when a web page is modified.
- *Content-Length* – the content length of a web page.
- *Content-MD5* – the MD5 encoding [6] of a web page.

Related cache control headers are:

- *If-Modified-Since* – a conditional GET if the content is modified since a timestamp.
- *ETag* – the *ETag* response-header field provides the current value of the entity tag for the requested variant.

In cache control, the header *If-Modified-Since* is based on the time information presented by *Last-Modified* when server determines a conditional get. Further analysis will show that *Last-Modified* header is not a reliable header for change detection all the time. Therefore, we ignore the *If-Modified-Since* header in *MetaDetector*.

We maintain a cache of all these related header values locally. The *MetaDetector* checks each of the related headers one-by-one in a certain order until it can successfully determine whether the web page is changed. The detection on each of the meta data returns one of three results: *CHANGED*, *UNCHANGED* and *UNKNOWN*. Figure 2.2 shows the control flow of *MetaDetector*.

The effectiveness and reliability of meta detection

Because web servers may present false meta data or even do not give the needed meta data, a detection of a single meta data, e.g. *Last-Modified*, may fail to determine whether a web page has been modified. We performed experiments on NYU web to examine the probability whether required meta data is presented by web servers. Additionally, for *Last-Modified* header field, if it is not the detection time and is within last three months, we mark it as *Good*. Table 2.1 shows the results of 159,731 successful detections on a set of 60,000 URLs in a three-day period where we attempt to detect each URL one time each day.

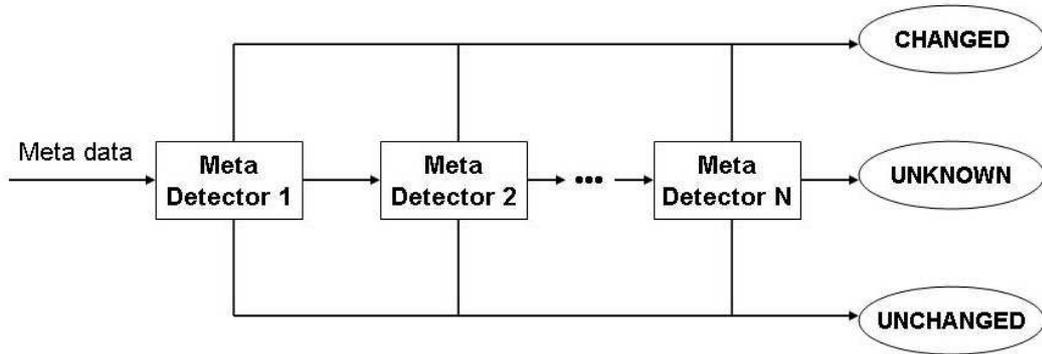


Figure 2.2: The control flow of *MetaDetector*.

Table 2.1: Presence of meta data

Meta data	Missing	Provided	Good
Last-Modified	33.8%	66.2%	11.3%
Content-Length	27.1%	72.9%	N/A
Content-MD5	100%	0.0%	N/A
ETag	34.1%	65.9%	N/A

We found no server in nyu.edu domain that presents *Content-MD5* header field. For *Last-Modified* header, we found most data is not quite reliable for change detection. The percentage given as **Provided** in Table 2.1 gives the coverage of each single meta data, which is not very satisfying if we choose only one of them for detection. The question is whether the coverage will be improved a lot using multiple meta data.

In order to study the coverage of multiple meta data, we select three meta data: *Last-Modified*, *Content-Length* and *ETag*, and study the coverage of their

arbitrary combination. The combined coverage shows that every HTTP message that provides “ETag” header also provides “Last-Modified” header; and every HTTP message that provides “Last-Modified” header also provides “Content-Length” header.

Such results are not encouraging at all. The combined coverage does not improve than individual presence. In most cases, a web server choose to provide most relevant meta data or choose not to provide at all.

The detection order of meta data

Each meta data has limitations in change detection. The *Last-Modified* header may provide false data, e.g. timestamp dynamically generated by web servers at runtime. The unchanged *Content-Length* does not guarantee that the content is 100% unchanged. To accomodate these limitations, we use the detection order given by Figure 2.3.

2.2.2 Level 2: *ContentDetector*

If meta detection fails and return a status of *UNKNOWN*, we download the web page. There are two methods for detecting changes here:

- *Content length detection* – We know that servers may not provide necessary meta data for *MetaDetector*. One case is that content length is not provided in HTTP header fields. For such cases, after downloading the document, we compare the actual content length of the current version with the length of local copy of the last modified version. If it is different, we report detection result as *CHANGED*, otherwise as *UNCHANGED*.

Detection order of Meta data

Last-Modified → Content-MD5 → ETag → Content-Length

Code:

```
IF Last-Modified is provided and is GOOD and is changed
    return CHANGED
ELSE IF Last-Modified is provided and is GOOD and not changed
    return UNCHANGED
ELSE IF Content-MD5 is provided and is changed
    return CHANGED
ELSE IF Content-MD5 is provided and is not changed
    return UNCHANGED
ELSE IF ETag is provided and is changed
    return CHANGED
ELSE IF ETag is provided and is not changed
    return UNCHANGED
ELSE IF Content-Length is provided and is changed
    return CHANGED
ELSE IF Content-Length is provided and is not changed
    return UNCHANGED
ELSE
    return UNKNOWN
```

Figure 2.3: Detection order of meta data.

It is rare that document is modified while the content length remains unchanged.

- *Content encoding detection* – This detection requires the computation of the encoding string of current version, which is the most expensive one during the detection phase. We compare the computed encoding string of the current version with the one of the last modified version stored in local database. If the encoding string is different, it returns status *CHANGED*, otherwise returns status *UNCHANGED*.

Why is *content encoding detection* needed after *content length detection*? First, it is more accurate than *content length detection* in order to identify *indexable changes* in web pages. Rather than encoding the content as a text document,

we use an encoding method for hypertext documents, called *HT-encoding*, presented Section 2.4.2. The advantage of *HT-encoding* is that it excludes non-indexable content, e.g. style sheets, images and white spaces, while preserves the structure of the hypertext documents. Second, it is less expensive than direct content comparison of current version and the last modified version. The encoding string for last version was computed when it was downloaded and is available for retrieval in database.

2.2.3 The effectiveness of change detector

A naive method for detecting changes is to download every page and detect changes offline. We compare the network traffic generated by our *MetaDetector* and the naive method here. The experiments on the NYU web show that the probability that *MetaDetector* reports *UNKNOWN* status is 0.27. Not counting the web pages that change, our crawler only need to download the content for 27.1% of URLs in our URL set. The average size of web pages in our data set is 10.5KB, and the average network traffic of HTTP request is 0.5KB per request. Our *MetaDetector* reduces the network traffic by 68.4% compared with the naive method. Consider that crawler for detecting changes runs at high frequency, the number of pages that are found changed during a short time is much smaller than the size of our URL set. For example, a 3-day detection on 60,000 URLs where each URL is detected one time each day only finds 1000-2000 changed pages in general. Counting the downloads of changed pages, the network traffic we need is still 67% less than the naive method.

Table 2.2 shows the fraction of changed documents that are found by differ-

ent detection methods. We find *content encoding* detection only reports that $0.412/0.794 = 51.9\%$ of those detected as *CHANGED* by *content length detection* contains indexable content changes.

Table 2.2: Percentage of changed documents found by different detection methods.

MetaDetection	ContentDetection	
	content length	content encoding
20.6%	79.4%	41.2%

2.3 Extracting web changes between different versions

Hypertext documents are structured by markups that can be used to build a document tree view. For example, the Document Object Model (DOM) [1] can be used to construct a document tree for a hypertext document. The document tree view of web documents carries more information than a text file and can be used to extract information more effectively than a text view. The question is: is it good for extracting web changes between different versions of web pages?

We have known that the amount of modification on existing web pages is significantly smaller than the content of web pages on average. When authors add or modify the content of web pages, it is very likely that only text is added or modified while the document tree structure and content layout in web browsers remains relatively static. Additionally, as more and more web authors tend to

maintain the structure of web pages as templates, the modified content tends to locate in the low levels of the document tree. Figure 2.4 is an example of

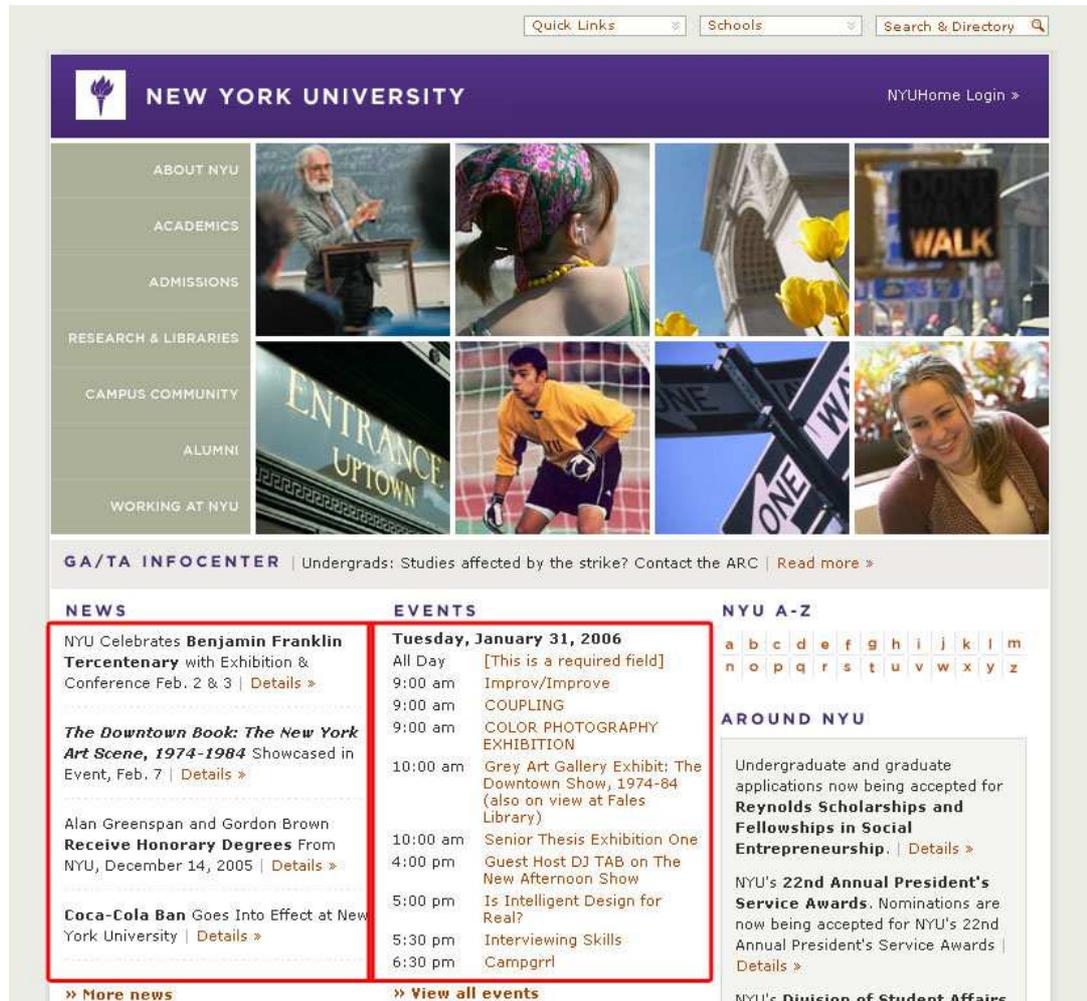


Figure 2.4: An example of modification areas on web pages.

web page using templates for layout. The blocks enclosed by the black boxes are usually used to present new information on this page. While looking at the HTML source code, we found the starting point of the red blocks is at depth 9 of the document tree and the red blocks spans 3 depths in the tree.

To accommodate such modification pattern, we use document tree to represent different versions of web pages in extracting web changes. Our algorithm is implemented in three steps:

- *Document tree construction*
- *Bottom-up tree encoding*
- *Top-down tree matching*

2.3.1 Document tree construction

We use the Document Object Model (DOM) [1] to build an HTML document tree. Unlike XML documents, the elements in HTML documents are not always presented in a nested style. There are two difficulties in building HTML document tree:

- According to HTML 4.01 Specification [3], the presence of end tag can be required or optional or forbidden. We define a tag that must have an end tag as *ReqTag*, a tag that can have optional end tag as *OptTag* and a tag that cannot have end tag as *ForbTab*. For examples, `<div>`, `<table>`, `<a>` are *ReqTags*, `<p>`, `` are *OptTags* and `
`, `<meta>` are *ForbTags*.
- Some formatting tags may be misaligned, such as ``. Figure 2.5 describes how misalignments can be found. Some HTML parsers, such as Microsoft Internet Explorer, can accommodate such misalignments to display the web page correctly.

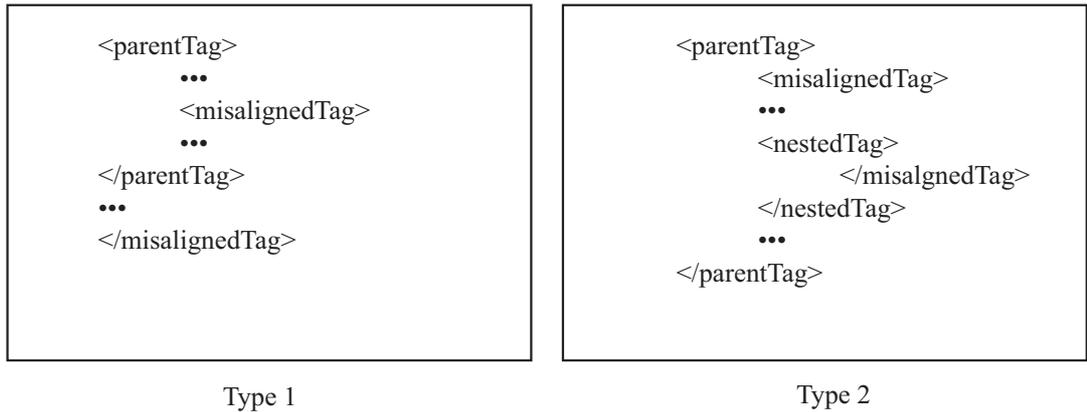
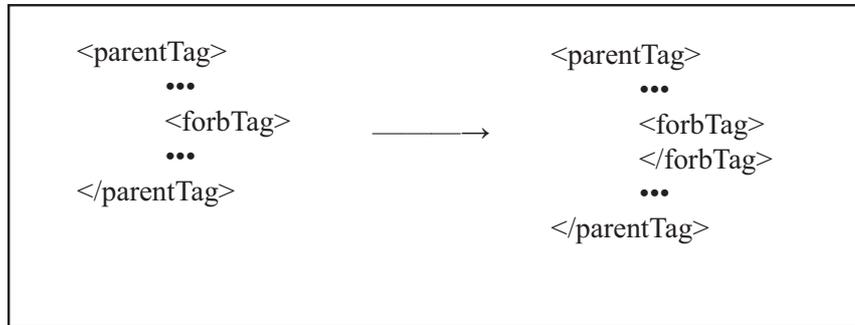


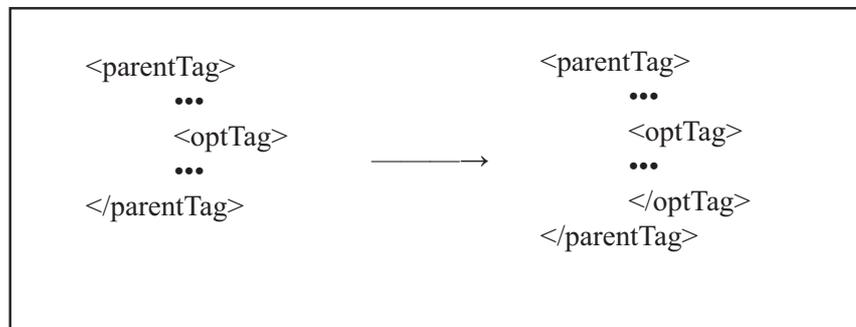
Figure 2.5: Misalignments of HTML tags.

The procedure of building a document tree is usually integrated in an HTML document parser. The parser parses the content and returns data when start tag or end tag or text data is found. The document tree constructor maintains a cursor starting at the root level, adds nodes and traverses through the tree during parsing. When an end tag is missing or misaligned, it adds an additional end tag to preserve the nested document structure. Specifically, if a *ForbTag* is found, it adds an end tag immediately after the start tag and maintains the cursor at the same depth (Figure 2.6 (a) shows the transformation for *ForbTag*); if an *OptTag* is found at depth d and the cursor moves to depth $d + 1$, but the end tag of *OptTag* is not found after the cursor has left depth $d + 1$, it adds an end tag (Figure 2.6 (b) shows the transformation when a single *OptTag* misses the end tag). There is a special case where multiple *OptTags* appear in the same subtree without end tags (e.g. `<parentTag> <p> ... <p> ... <p> ... <parentTag>`). Rather than placing them in a nested sub-tree, the parser places them at the same depth in the sub-tree of `<parentTag>` (Figure 2.6 (c) shows the transformation when multiple same *OptTags* are in the same level and miss

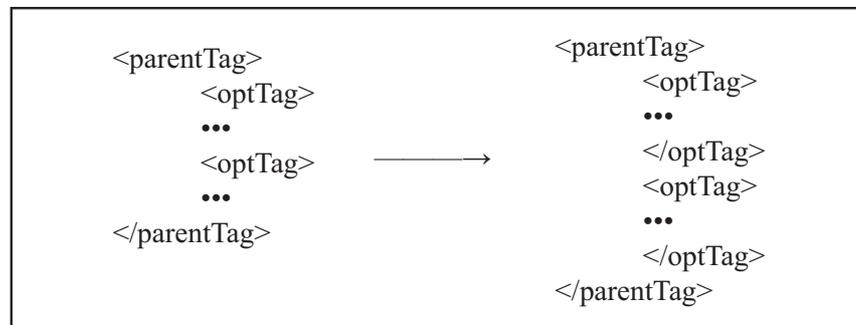
end tags). In the layout of multiple same *OptTags* such as $\langle p \rangle$, flat structure is preferred over nested structure.



(a)



(b)



(c)

Figure 2.6: Tag structure transformation for *OptTags* and *ForbTags*.

The misalignment problem is more difficult. Theoretically, there is no unique tree structure that can match an arbitrary misalignment. By examining the web data we collected from NYU web, we found most misalignments come from `` tag. Other misaligned tags exist but are rare, e.g. `<form>`. For Type 1 misalignment given in Figure 2.5, we move the end tag out of the sub-tree to the same depth of start tag. For Type 2 misalignment, we add an end tag at the same depth of start tag but ignore the actual end tag.

Listing 2.1 gives the algorithm for constructing document tree using an event driven document parser. In our tree construction, we exclude two kinds of HTML tags and their nested content: script (`<script>`) and style sheet (`<style>`). A script is a piece of program written in HTML document to add additional functions. They are not indexable and not relevant to the document structure. Style sheets are often used to give customized layout of documents in web browsers. The nested text in style sheet often contains dynamic content such as advertisement image banner. We do not want to extract such changes.

Listing 2.1: Tree Construction Algorithm using event-driven parser

```

1 Initialize :
2   // document tree
3   DocTree = RootNode
4   // parent node for current node
5   ParentNode = RootNode
6   depth = 1 // current tree depth
7   // a list storing nodes from root to current node
8   DocPath = (RootNode)
9   // a list of current pending misalign tag
10  MisalignTags = (empty)
11
12 BuildDocTree(InputFile input)
13 {
```

```

14 // setup parser
15 Register foundStartTag(), foundEndTag() and foundText()
16 as parsing handlers
17 // parsing
18 parse document input
19 return DocTree
20 }
21
22 foundStartTag(Tag t)
23 {
24     // create a new node
25     Node n = createElementNode(t)
26     if t is ForbTag {
27         Add n to the child list of ParentNode
28     }
29     else if t is OptTag {
30         if the tag of ParentNode is the same OptTag {
31             // finish last OptNode
32             ParentNode = DocPath[--depth]
33             Add n to the child list of ParentNode
34             DocPath[++depth]=n
35         }
36         else {
37             Add n to the child list of ParentNode
38             // move to the next level
39             ParentNode = n
40             Add n to the end of DocPath
41             depth++
42         }
43     }
44     else { // ReqTag
45         Add n to the child list of ParentNode
46         // move to the next level

```

```

47     ParentNode = n
48     Add n to the end of DocPath
49     depth++
50 }
51 }
52
53 foundEndTag(Tag t)
54 {
55     if t is in MisalignTags {
56         // process type 1 misalignment
57         remove t from MisalignTags
58         return
59     }
60     find the index i of last node in DocPath of tag t
61     if ( i == depth-1 ) {
62         // finish a normal node
63         ParentNode = DocPath[--depth]
64         // process type 2 misalignment
65         if MisalignTags is not empty, update DocPath
66     }
67     else if ( i < depth - 1 ) {
68         for each node j between i and depth in DocPath {
69             if the tag of DocPath[j] is ReqTag {
70                 // type 1 misalignment
71                 add tag of DocPath[j] to MisalignTags
72                 depth = i - 1
73             }
74         }
75         // type 2 misalignment
76         Add t to the end of MisalignTags
77     }
78 }
79

```

```
80 foundText(Text t)
81 {
82     Node n = createTextNode(t)
83     Add n to the child list of ParentNode
84 }
```

The nodes in the tree we construct carry several additional attributes not presented in the original document:

- The location of the tag in the original document.
- The content length represented by the sub-tree.
- The encoding string of the sub-tree. The encoding method is described in Section 2.3.2.

The first two attributes are preserved for information segmentation and evaluation as we will discuss later. The third one is preserved for performing tree matching and information extraction as described in Section 2.3.3. The encoding string of the root node is also used by *ContentDetector* to detect whether a document has been changed.

2.3.2 A bottom-up tree encoding for hypertext documents: *HT-encoding*

In order to characterize changes between two versions of a web document, we need to match nodes between the corresponding document trees. Since modifications on web pages are often located on low levels of the tree and many sub-trees remain unchanged, an efficient comparison involves encoding each sub-tree, removing identical sub-trees then matching the rest. Here we present a Hypertext

Tree encoding (HT-encoding) algorithm to encode the document tree level-by-level. For each sub-tree in the document tree, the content and structure are encoded as a string, which is stored in the root node of the sub-tree. If two sub-trees share the same encoding string, the content and the tree structure must be the same. However, the attributes attached to tags are not encoded; thus if the formatting given by tag attributes or the source link of images are modified, the encoding string is not changed. For text information retrieval, such changes are not considered as indexable. Listing ?? shows the HT-encoding algorithm. The main structure of the algorithm is a Depth-First-Search (DFS) traversal over the tree. The procedure *HT-Encoding* takes an argument of the root node of a sub-tree, recursively computes the *HT-encoding* string of the immediate children of the root node. We use MD5 [6] algorithm to encode text string. On return, it concatenates its tag string and all encoding strings of its children and computes the MD5 encoding. The encoding string is added as a node attribute.

Listing 2.2: The HT-encoding algorithm for document tree

```

1 Given :
2   a DocTree — a document tree
3 On return :
4   For each node in the tree , a node attribute of
5   encoding string is added.
6
7 HT-Encoding (DocTree t)
8 {
9   if t is a TextNode {
10      // encode text data
11      Add attribute ('encoding' , MD5(t.textData)) to t
12   }
13   else {

```

```

14     // processing an element node
15     String encodingStr = t.tagName();
16     // encode all sub-trees
17     for each child t_child node of t
18     {
19         HT_encoding(t_child)
20         Append t_child.encoding to the end of encodingStr
21     }
22     // encode root node
23     Add attribute ('encoding', MD5(encodingStr)) to t
24 }
25 }

```

2.3.3 An efficient algorithm for tree matching

General purpose tree matching algorithms for ordered or unordered trees are computational expensive. They often use tree edit distance to measure how different two trees are. For matching ordered trees, the best algorithm is given by Zhang and Shasha [67] which has the complexity of the sub-cubical of the size of trees. Matching unordered trees is an NP-complete problem in general [68]. Tree edit distance has been used to detect and extract changes between XML documents [62, 30]. All existing algorithms are of at least polynomial complexity, which are inappropriate for processing mass web data. Here we present a restricted tree matching algorithm for extracting changes from different versions of web page, which has linear scalability.

Given the encoded trees of two versions of a web page, we address our algorithm for extracting the changes from the latter one over the other. Our algorithm utilizes the following properties of web changes:

- While changes may be posted at high frequency, the overall structure of the web document tend to be static for a long time.
- When a block of content is modified, the starting depth in the document tree tend to be static for a long time.
- The tag of the parent node of a changing block is relatively static too.

Our algorithm compares two document trees level-by-level, removes common sub-trees while matches the rest through the tag and second-level tree structure. Listing 2.3 shows the algorithm. The output tree is a “shrunked” tree of the new version, which contains the changes as well as the tree structure of the changes. We call it *ChangeTree*.

Listing 2.3: The algorithm for extracting changes from two versions of web documents

```

1 Given :
2   two document trees , T1, T2
3 On return :
4   a 'shrunked' document tree t1 containing the changed nodes
5   and the corresponding tree structure upto the top root node
6
7 MatchTree ( DocTree T1, DocTree T2)
8 {
9   for each child node C1 of T1 {
10      // 1st round: remove common sub-trees
11      if T2 has a child C2 with the same HT-encoding as C1 {
12          remove C1 from the child list of T1
13          remove C2 from the child list of T2
14      }
15      // 2nd round: matching changed nodes

```

```

16     Group child nodes of T1, T2 by tag name
17     for each matching tag group G1 of T1 and G2 of T2 {
18         for each C1 in G1 {
19             Find the best matching node C2 in G2
20             MatchTree(C1, C2)
21         }
22     }
23 }
24 return T1
25 }

```

The best matching node at line 20 is determined by the following function

```

26 CountMatches (DocTree C1, DocTree C2)
27 {
28     count = 0
29     for each child node CC1 of C1 {
30         if C2 has non-matched child node CC2
31         with the same HT-encoding as CC1
32             count++
33         else if C2 has non-matched child node CC2
34         with the same tag name as CC1
35             count+=0.5
36     }
37     return count;
38 }

```

During the process of extraction, each sub-tree in the original document tree may have one of three outcomes: completely removed if identical sub-tree is found; remaining intact if no match is found; “shrunk” if matched. There are a few limitations of our algorithm: if the document tree is reorganized and

common sub-trees are placed at different depth, we cannot match them; if the parent tags of identical sub-trees are changed, we cannot match them either. Both situations are rare in the modifications of web documents. Matching trees using tree edit distance can tolerate such situations to some degree. In particular, it is widely used to match small tree patterns into a large tree. However, most algorithms of tree edit distance have high computation complexity which is not proper for processing mass data such as the Web. Our algorithm runs in linear cost and is suitable for the extraction task we addressed.

2.3.4 Complexity of algorithms

Denote N as the total number of nodes in the document tree, the number of children of a node as C , and the depth of the document tree as D . If the tree is not significantly unbalanced, the depth D is in the order of $O(\log_C(N))$.

The document tree construction algorithm runs in linear time for a strictly nested HTML document since each tag is processed only twice (start tag and end tag). Misalignments may introduce minor additional cost.

The *HT-encoding* algorithm navigates the document tree only once thus it has linear scalability. The complexity is of the order of $O(\eta N)$ where η is the average cost for computing *MD5* string.

In order to analyze the cost of our tree matching algorithm, we denote p as the fraction of nodes with identical encoding strings between two trees, q as the fraction of matched nodes and r as the fraction of non-matched nodes, where $p+q+r = 1$. Denote K as the number of non-overlapping sub-trees whose nodes have common encoding strings between two document trees. From the results in Appendix A, we get $K = O(\frac{pN}{D}) = O(\frac{pN}{\log_C(N)})$. The cost of removing these trees

is in the order of $O(\frac{pCN}{\log_C(N)})$. The similar result holds for non-matched nodes, which is $O(\frac{rCN}{\log_C(N)})$. For matched nodes, the computation cost is of $O(C^2qN)$. Hence the total complexity of the algorithm for the tree matching algorithm is:

$$\text{Cost} = O(\frac{pCN}{\log_C(N)}) + O(C^2qN) + O(\frac{rCN}{\log_C(N)}) \quad (2.1)$$

The analysis above makes an assumption that the number of children for non-leaf node is a constant. However, the parameter C varies for different nodes. The actual complexity of the algorithm is determined by the following expected values: $E(C)$, $E(C^2)$ and $E(N)$. We cached about 220,000 web pages in the domain of nyu.edu. We collected the statistics of these pages and got the following results:

- The average number of tag nodes in the document tree is 176.0 and the average number of text nodes is 98.5. Thus $E(N) = 274.5$.
- The average depth of document tree is $E(D) = 11.5$.
- The average number of children of non-leaf node is $E(C) = 1.75$. The expected depth of a balanced tree is $\log_{1.75} 274.5 = 10.0$, which is smaller but not greatly smaller than the actual average depth.
- The average of the square of number of children of non-leaf node is $E(C^2) = 22.8$. This number is not small when computing the complexity of finding the matched nodes. However, our algorithm groups nodes of same tag names before pairwise matching. The actual complexity for finding matched nodes is determined by $qN \cdot E(\sum_i C_i^2)$, where C_i is the number of child nodes of tag group i . Our experiments show that $E(\sum_i C_i^2) = 7.75$.

Our algorithm is used to process large number of web pages. We are more interested in the average cost than the cost of a single case. Therefore, we

consider the coefficients related to C as constant factors. Practically, the overall cost is linear in the size of document tree. However, depending on different modification patterns, the cost varies and may be lower than linear.

- When the structure of the document tree remains relatively static while only small portions of content are modified, the cost is dominated by $O(\frac{pCN}{\log_C(N)})$, which is less than linear.
- When the structure of the document tree remains relatively static while most content is modified, which is less likely, the cost is dominated by $O(C^2qN)$.
- When the structure of the document tree is modified thoroughly, the cost is dominated by $O(\frac{rCN}{\log_C(N)})$, less than linear.

Optimization of tree matching algorithm

When the fraction of matched nodes is significant and the number of children of a node is big, pairwise matching is costly. The algorithm can be optimized using certain data structure. To simplify our analysis, we consider that all child nodes have same tag name and are nodes to be matched. Let $c_i, c'_{i'}$, $i = 1, 2, \dots, k$ and $i' = 1, 2, \dots, k'$, be current level nodes in the two matched subtrees; and $c_{ij}, c'_{i'j'}$, $i = 1, 2, \dots, k$ and $j = 1, 2, \dots, k_i, j' = 1, 2, \dots, k'_{i'}$, are child nodes of next level. The task is to match nodes between $c_i, c'_{i'}$, where $i = 1, 2, \dots, k$ and $i' = 1, 2, \dots, k'$. First, we store all second level nodes, $c'_{i'j'}$, in a hash table where the hash key is the encoding string of $c'_{i'j'}$. Second, for each node c_{ij} , we lookup the encoding string in the hash table. If the encoding string is the same as $c'_{i'j'}$, we increment the count of matches, $match[i, i']$, by 1. Third, for each i , we match c_i to c'_i where $match[i, l] = \max_{\text{for each non-matched } c'_l} (match[i, i'])$.

The cost of the first step is linear to $\Sigma_{i'}k'_{i'} \sim C^2$; the cost for the second step is linear to $\Sigma_i k_i \sim C^2$; the cost for the third step is $\Sigma_i k' \sim C^2$. The average cost for matching each node is $O(\frac{C^2}{C}) = O(C)$.

2.3.5 Information segmentation

Web changes may be scattered within a web document and may provide information on different topics. We use the location of changes and HTML separation tags to divide changes into multiple pieces of segments. There are two cases need to be studied: separated changes and continuous changes.

If two neighboring sub-trees in the *ChangeTree* are physically remote in the original web document, that is, many common sub-trees between them are removed during the process of change extraction, then we put them into different segments.

If two neighboring sub-trees in the *ChangeTree* are continuous in the original web document, we use HTML tags to separate them and put into different segments. Table 2.3 shows the tags we use. In addition, we also check the text length separated by these tags. If the length does not exceed a threshold, we group neighboring segments into a single segment. All these segments are stored in database as *New Information Fragments (NIF)* for searching.

2.3.6 Demo

We illustrate the process of change extraction using two versions of the home page of New York Times. Figure 2.7 shows the screen shots of the page at 5pm, Jan 31, 2006 and 5pm, Feb 31, 2006 respectively. We process these two versions using the algorithms we have studied: building document tree, encoding

Table 2.3: HTML tags for separating content

Parent tag/child tag	Function
<DIV>	Create block-level structure of content
<P>	Separate paragraphs
Double 	Separate content using double line breaks
<TABLE>/<TR>	Create table layout
,/	Create list layout

the tree, extracting changes from the second version over the first one, and dividing *ChangeTree* into segments. Figure 2.8 shows the result, which is the serialized view of the extracted *ChangeTree*. Some additional content extracted cannot be displayed due to space limitation, such as some new headlines. We find the extracted content do succeed in identifying and segmenting the new information on the page. Surrounding text, including page banner, navigational links, image advertisements and utility links are removed successfully. In the phase of extracting web changes, we preserve the line breaks between sub-trees in the *ChangeTree*, which help preserve the layout of the changes. The information segments generated by computer are shown in Figure 2.9. Compared with the layout of the information in Figure 2.8, such segmentation is reliable.



Version 1
<http://www.nytimes.com/>
 Jan 31, 5pm, 2006

Version 2
<http://www.nytimes.com/>
 Feb 2nd, 5pm, 2006

Figure 2.7: Two versions of the home page of New York Times.

[Ohio Congressman Wins Majority Leader Race, Replacing DeLay](#)

By CARL HULSE and DAVID STOUT 4:11 PM ET

John Boehner defeated Roy Blunt in a stunning upset signalling many House Republicans' concerns about recent lobbying scandals.

[Democrats Seek Prosecutor in Lobbying Case](#)

[Democrats and Bush Aides Spar in Senate Over U.S. Spying](#)

By SCOTT SHANE 4:45 PM ET

An annual hearing on national security threats was overtaken by partisan debate about the domestic surveillance program.

[Senate Panel Rebuffed on Spying Documents](#)

[Iran Vows to End Cooperation if Nuclear Case Goes Forward](#)

By ELAINE SCIOLINO 1:21 PM ET

The threat means that the International Atomic Energy Agency would lose access to key sites and installations.

[Text: Iran's Message to the I.A.E.A. \(pdf\)](#)

SUPER BOWL XL

[Two Steelers Made It in Detroit](#)

With his parents' help, [Jerome Bettis](#) left a broken neighborhood behind. [Larry Foote](#) was unafraid to face his fatherhood early on. [Complete Coverage](#)



Doug Mills/ The New York Times
Representative John Boehner spoke to reporters after he was chosen to succeed Tom DeLay as the House Republican majority leader.

ARTS

[Met Sending Vase to Italy, Ending 30-Year Dispute](#)

INTERNATIONAL

[Firestorm Over Cartoon Gains Momentum](#)

[Brooks: Nation of the Future](#)
[Herbert: American Obsession](#)

[The Opinionator: Next DeLay](#)
[? Editorial: Hamas at the Helm](#)

Figure 2.8: The changed content extracted.

UPDATED THURSDAY, FEBRUARY 2, 2006 5:01 PM ET

Ohio Congressman Wins Majority

Leader Race, Replacing DeLay By CARL HULSE and DAVID STOUT 4:11 PM ET John Boehner defeated Roy Blunt in a stunning upset signalling many House Republicans' concerns about recent lobbying scandals. Democrats Seek Prosecutor in Lobbying Case

Democrats and Bush Aides Spar in Senate Over U.S. Spying By SCOTT SHANE 4:45 PM ET An annual hearing on national security threats was overtaken by partisan debate about the domestic surveillance program. Senate Panel Rebuffed on Spying Documents

Iran Vows to End Cooperation if Nuclear Case Goes Forward By ELAINE SCIOLINO 1:21 PM ET The threat means that the International Atomic Energy Agency would lose access to key sites and installations. Text: Iran's Message to the I.A.E.A. (pdf)

SUPER BOWL XL Two Steelers Made It in Detroit With his parents' help, Jerome Bettis left a broken neighborhood behind. Larry Foote was unafraid to face his fatherhood early on. Complete Coverage ... Doug Mills/ The New York Times Representative John Boehner spoke to reporters after he was chosen to succeed Tom DeLay as the House Republican majority leader.

ARTS Met Sending Vase to Italy, Ending 30-Year Dispute

INTERNATIONAL Firestorm Over Cartoon Gains Momentum

Brooks: Nation of the Future Herbert: American Obsession The Opinionator: Next DeLay ? Editorial: Hamas at the Helm

Klinkenborg: Early Spring Warner: Peanuts and Pediatricians

Rhoden: Fighting for Past Players Kristof: Bob Woodruff

Figure 2.9: Information segments generated.

2.4 A framework of evaluating web changes

Web pages indexed by public search engines are ranked using the popularity of web pages in the global web and the relevance to user queries. For the evaluation of web changes, these metrics are not enough and may not be good candidates to evaluate web changes. In order to evaluate the *New Information Fragments (NIF)* in the local database, we combine the following metrics to produce an unified ranking: popularity, content-based evaluation and evolution. Popularity ranking and quality ranking produce the static ranking scores of NIFs. However,

the evolution ranking evaluates the changes over time. Figure 2.10 shows the framework of our ranking strategy.

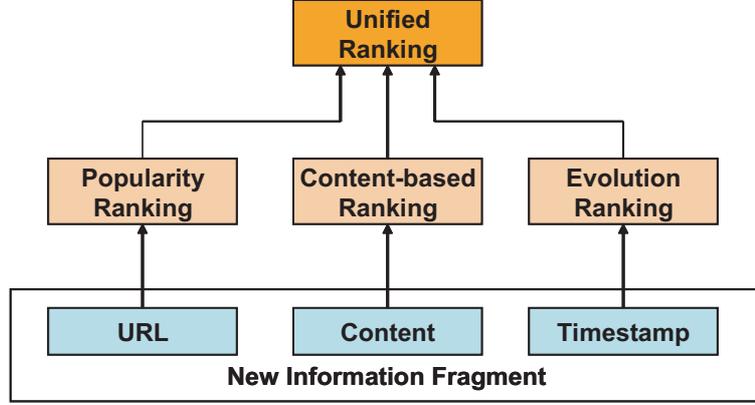


Figure 2.10: The framework of ranking web changes.

2.4.1 Popularity ranking

The popularity ranking uses the link structure of URLs to infer which pages are important in the web graph. The best known method for doing this is the PageRank algorithm [55] in Google. In this section, we first review the PageRank algorithm, then we present an improved algorithm that modifies the underlying stochastic process of PageRank to produce better ranking results.

An introduction to PageRank algorithm

A web graph of n pages can be represented using an n -by- n adjacency matrix A , where element $A(i, j)$ is 1 if page i links to page j , and 0 otherwise. The PageRank algorithm first constructs a probability transition matrix M by normalizing each row of adjacency matrix A to sum to 1. The idea of this algorithm

corresponds to random walk within a graph. When a person is at page i , with probability $(1 - \epsilon)$, it uniformly picks a link from this page and transits to the target page of this link. With probability ϵ , it jumps to any other page in the web with uniform probability. The transition matrix T is

$$T = \epsilon U + (1 - \epsilon)M \tag{2.2}$$

where U is an n -by- n matrix of uniform transition probabilities having $U_{ij} = 1/n$ for all $1 \leq i, j \leq n$. The PageRank score for a page P is the probability that the random walker visits P on a very long walk. More formally, the vector of PageRank scores p is then defined to be the stationary distribution of the Markov chain corresponding to the random walk model as described. It satisfies:

$$(\epsilon U + (1 - \epsilon)M)^T p = p \tag{2.3}$$

where p is the vector of importance scores of PageRank. The ϵ here is typically chosen as 0.15. The computation of PageRank is recursive, on-line and resource inexpensive [55, 39]. Equation 2.3 shows the score vector p is an eigenvector of matrix $(\epsilon U + (1 - \epsilon)M)^T$. Practically the score vector is computed via power iterations as follows:

$$p_{t+1} = (\epsilon U + (1 - \epsilon)M)^T p_t$$

The convergence of this iteration is fast. Page and Brin [55] compute PageRank on a web graph containing 75 million distinct URLs and 322 million links and it converges in 52 iterations with satisfying accuracy. They also show that the computation of PageRank scales well that the number of iterations needed for convergence is logarithmic in the size of web graph.

An improved ranking algorithm based on web graph information

Link based algorithms for ranking web pages, e.g. [55, 43], often follow a stochastic process over the web graph. In the computational models of these stochastic processes, if there exists a path from node i to node j and also a path from node j to node i in the selected web graph, then theoretically, the rank value of node i will contribute to the rank value of node j and vice versa. We call this effect as *Circular Contribution*. The magnitude of such contribution depends on the actual web graph structure and the actual transition matrices in the algorithms. In general, Circular Contribution is one important computation mechanism in stochastic models and can help produce good ranking. But it can cause over-ranking problem if the web graph contains tightly linked sub-collections. We improve PageRank algorithm by computing the backward distance of linked web pages and biasing the random walk with minimum backward distance [63].

Denote G as the web graph of a set of web pages. The minimum backward distance (denoted as *MinBD* for short) is defined as follows:

Definition 3. *If there is a (i, j) from page i to page j , the minimum backward distance is the length of the shortest path from page j to i in the graph of G .*

An efficient algorithm for computing *MinBD* is presented in Appendix B. In *Circular Contribution*, we know the mutual contribution of two nodes is affected by the length of the path between them. The smaller the length is, the greater the contribution is. Thus, for each link $i \rightarrow j$, the backward contribution of node j to node i is dominated by the minimum backward distance from node j to node i in the web graph. For each link $i \rightarrow j$ the mathematical representation

of the evaluation is

$$v_{ij} = \begin{cases} \frac{f(\Omega)}{C_i} & \text{if } MinBD_{ij} \geq \Omega \\ \frac{f(MinBD_{ij})}{C_i} & \text{otherwise} \end{cases} \quad (2.4)$$

where Ω is a threshold, C_i is the normalization coefficient such that the sum of v_{ij} is 1 respect to index j , $f : R \rightarrow R$ is an increasing function satisfying $f(0) \geq 0$.

Then we can modify the underline stochastic process of PageRank algorithm using the above link evaluation. The new transition matrix \tilde{M} is

$$\left(\tilde{M}_{ij}\right) = (v_{ij}) \quad (2.5)$$

Since the link values have been normalized, \tilde{M} is a well formed transition matrix. Replace the original matrix M in Eq. 2.3 with \tilde{M} , new ranking values can be computed.

Let's consider the random walk guided by this link evaluation. Suppose the web surfer can "see" far more than the neighbors at any page. An efficient web walk strategy would be to explore the web as much as possible and not go back to the visited local pages frequently. When determining the next transition, the surfer preferentially jump to the linked pages with higher $MinBD$. Consequently, the probability that the surfer goes back to the current node is smaller than uniform transition selection. Furthermore, as those links with small $MinBD$ are evaluated less, the effect of *Circular Contribution* is decreased for tightly linked sub-collection whose web graph contains many short cycles.

In order to evaluate our algorithm, we manually selected six subcollections of the NYU web that are tightly connected but not particularly important, including message boards, large projects, etc. The average ratio of the number

of links within a collection as compared to the number of links entering and leaving collection is 74.24 : 1. An a web graph of about 100,000 web pages is studied. We compare the ranks between PageRank and the ones given by our algorithm to evaluate the effectiveness of our algorithm in reducing the overranking problem. The precise algorithms use the following configurations: in PageRank algorithm we set $\epsilon = 0.15$ as Page et al. [55] does; in link evaluation we use f , $f(x) = \sqrt{x}$ in Eq. 2.4 to scale *MinBD* values. We set the threshold $\Omega = 15$ considering that the backward distance is large enough when *MinBD* is greater than 15. Let Φ be a subcollection of local aggregation, $R_1(i)$ and $R_2(i)$ be the ranks of page $i \in \Phi$ in two different ranking algorithms². The evaluation methods are defined as:

- Average rank difference ADiff:

$$\text{ADiff}(\Phi, R_2, R_1) = \frac{\sum_{i \in \Phi} (R_2(i) - R_1(i))}{\text{Size}(\Phi)}$$

- Highest rank difference HDiff:

$$\text{HDiff}(\Phi, R_2, R_1) = \min_{i \in \Phi} R_2(i) - \min_{i \in \Phi} R_1(i)$$

Let R_2 be the rank given by improved algorithm and R_1 be the rank given by PageRank algorithm. The average rank given by PageRank algorithm of all sub-collections studied are within the top 27% places of the whole NYU web. The top rank of each sub-collection is high too. The worst case is that 4 directory

²The rank number is an increasing integer as the importance of web pages decrease. Thus, the rank of highest score given by a ranking algorithm has a rank number of 1. If the results in ADiff or HDiff is positive, it shows R_2 has a rank distribution lower than R_1 . For sub-collections tightly linked inside and should not get high global importance, the positive numbers given by the evaluation methods show R_2 is superior than R_1 .

pages of a message board is ranked within the top 20 positions. Experiments show that the average rank difference measured by ADiff is 3082.3 and top rank difference measured by HDiff is 60.7 for all sub-collections selected on average.

2.4.2 Content-based ranking

In evaluating the content of new information carried by web changes, we have two considerations: how much information carried and how timely that the information is. The amount of information can be evaluated simply from the length of the NIFs. In order to evaluate how timely web changes are from the content of web changes, we need to know what kind of words can be good indicators of new information.

We followed top 60,000 pages ranked by popularity from the Web of New York University everyday from March 2005 to November 2005 and collected the complete change history. We count the word frequency of the web changes found as well as the word frequency of all web pages. After removing the stop words, general verbs and too common words that are unrelated to new information, we select the top 100 words. Then we divide the top 100 words into five groups: time information words, time related words, university related words, popular topic words and the rest go to misc words. The results are shown in Appendix C. It shows that web changes are more likely to contain time or time related information. This finding is reasonable because web changes often present news or events which are closely related to time.

News or event information in web changes often have short lifetime. Such information also has more significance when we present web changes. Therefore, we count the appearances of time related words in NIFs as a metric of quality

ranking. Let L be the length of an NIF; and K be the number of time related words in the NIF. The quality score is defined as:

$$S_{content} = w_1L + w_2K \quad (2.6)$$

where w_1, w_2 are weights determining the scale of quality ranking score.

2.4.3 Structured evolution ranking

Popularity ranking and content ranking can produce static ranking of information. However, the importance of each web change decreases as time goes on. The evolution ranking reflects this decrease over time. We evaluate the novelty of web changes using an exponential function. Let S_0 be the score of static ranking. The online evaluation of new information at query-time goes as follows

$$S = S_0 \cdot e^{-\alpha t} \quad (2.7)$$

where α is an adjustable parameter to determine how fast the score decreases. $t = 0$ corresponds to the timestamp when the modification is made or when the change is detected if the former is not available.

How should α be set? A simple solution is using a fixed value of α for all. However, the lifetime of different *NIFs* are very different.

Definition 4. *The lifetime of a NIF is the time period between the time of its creation and the time of its removal on a web page.*

When a NIF has been removed from the web, we consider the information it carries is obsolete and of little importance. Denote the average lifetime of web changes on a single page as T_c , the ranking score of a web change at its creation as S_0 and the score when it is removed as ϵS_0 (our configuration of ϵ is

0.2, that is the importance of a web change decreases to $\frac{1}{5}$ of its original score at the time of its removal). The parameter α satisfies:

$$S_0 \cdot e^{-\alpha T_c} = \epsilon S_0$$

Therefore

$$\alpha = -\frac{\log(\epsilon)}{T_c} \quad (2.8)$$

Direct estimation of T_c by observing whether a change is removed on a page is difficult. We use the average lifetime of versions of a web page to estimate the average lifetime of web changes on the page.

Definition 5. *The lifetime of a version of a web document is the time period between the time of its creation, denoted as t_0 , and the first time when indexable content is added or removed after t_0 .*

For dynamically generated pages whose last modified time is always request time while the content is the same, we do not consider a new version is created. Denote the average lifetime of versions for a single web page as T_v . From Appendix D, we get the estimation of the average lifetime of web changes as

$$T_c = 2T_v$$

Furthermore, we observed that the update of many pages are very passive. Even a version containing a web change can exist for a long time, the information it carries can still be obsolete. Hence, we put a threshold, Θ , on the average lifetime of versions. Then the model for choosing α is given as

$$\alpha = -\frac{\log(\epsilon)}{2 \min(T_v, \Theta)} \quad (2.9)$$

The average lifetime of versions is the inverse of change frequency, which can be estimated from the detection history. The detailed coverage of how to estimate change frequency is given in Chapter 4. Figure 2.11 shows the results for configuration $\epsilon = 0.2, \Theta = 10$.

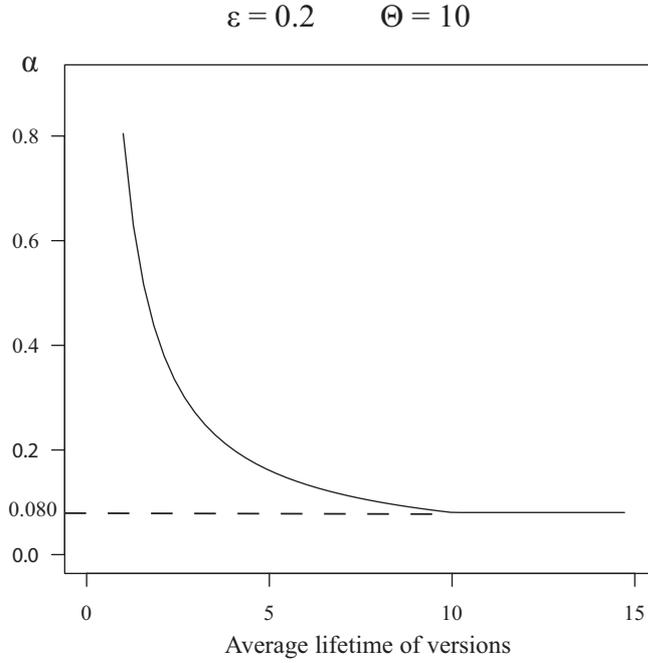


Figure 2.11: The selection of α .

2.4.4 Unified ranking

Given the methods and ranking scores of popularity ranking, quality ranking and evolution ranking, the formulation of unified ranking score is straightforward. Denote S_p as the normalized score of popularity ranking, S_c as the normalized score of content-based ranking and the α of evolution ranking. The

unified ranking score of *New Information Fragments* at query time is

$$S = (S_p + S_c) \cdot e^{-\alpha(t-t_0)} \quad (2.10)$$

where t_0 is the creation time of the web change.

We choose the sum of S_p and S_c rather than the product as static score. S_p measures the importance of the page that contains the change. We consider it as a base score when we evaluate changes. The quality score can be considered as an adjustment to the base score. Such combination can well distinguish the significance of multiple changes of the same page at the same time. For changes on different pages, which is evaluated more than one another is determined by the configuration of the numerical scale between S_p and S_c .

2.5 Summary

In summary, we analyzed the issues of collecting and presenting new information from web content changes and identified three central, inter-related problems: how changed pages can be detected, how changes of different versions of web pages can be extracted and how changes can be evaluated. We developed techniques and algorithms that can be used to solve these problems. We proposed a two-level change detector to detect candidate pages of web changes, effective algorithms to construct HTML document trees, encode document trees and extract changes between different versions of web pages, an improved algorithm for ranking popularity of web pages, and a framework of how to evaluate web changes. Experimental evaluations show these solutions are effective in collecting and presenting new information over the web.

Chapter 3

Application: Web Daily News Assistant

3.1 Design

In this Chapter, we present a brand new searching application, Web Daily News Assistant (WebDNA): Finding What's New on Your Web, which can assist people to automatically find what is new on the community Web. Consider the following situation when people use the community Web:

A community has changed their work schedule and got it updated on the web. A person in this community needs to be notified that the work schedule has been changed. If the web is the only media to deliver such information, in order to get new updates on time, he must visit the web page posting the work schedule frequently. This is time consuming and erroneous because he never knows whether or when the schedule will be changed. Furthermore, the situation gets worse if he has to keep in mind a lot of things similar.

Our goal here is to develop a search utility that can assist people in the similar situations to get valuable new information. There is an XML technology, Really Simple Syndication (RSS), which is a lightweight XML format for syndicating news and the content of news-like sites, including major news sites, news-oriented community sites and personal weblogs. It is a server based technology which requires the web site to distribute the RSS content. Here we consider a different approach, which uses web crawlers to find the changes and news, and provides a search capacity to user. Our application is specifically designed for use on the scale of a community web, such as a web site or a local web domain. Within such a scale, we can update the index much more frequently than general public search engines to the rate of a couple of times per day. Furthermore, our application is particularly interested in retrieving what has been changed recently. WebDNA is deployed on the New York University web site. It has the following functionalities:

- Presenting a news digest university wide.
- Presenting news digests for different departments.
- Presenting The changing history of a single page.
- Full-text keyword search on web changes.
- Managing and presenting the summary of changes on a user defined URL set.

Figure 3.1 gives a sample output of this system.



Figure 3.1: The user interface of WebDNA.

In designing this application, we have two major considerations here: one is *freshness*, the other is *quality*. Although community webs are small compared with the global Web, maintaining a fresh index of such web is still very challenging. The size of community web is typically of the scale $10^4 \sim 10^6$ pages. For example, from our crawling records, we found there are roughly 600,000 different URLs in the Web of New York University (nyu.edu) and about 200,000 of them point to retrievable HTML documents. From [41] we know, Internet performance is the major bottleneck of crawler-like applications. For community web search engines, the performance of web servers becomes bottleneck too. Like the World Wide Web, a community web is a distributed system if there are more than one web servers running, but the number of hosts is of course very small compared to the World Wide Web. In many community webs, a large fraction of the web pages are delivered by a single host, e.g. www.nyu.edu in

nyu.edu domain. While updating the local data, the crawler cannot avoid downloading from the same host many times in a short time. The capacity of such server will finally limit the speed of downloading. Furthermore, crawlers must observe a politeness policy to avoid overloading web hosts, especially when such web hosts serve a large community population. Typically, only several requests per second for each server are allowed. If the crawler runs at an aggressive downloading rate, the server may block the IP address of the crawling machine. Large-scale search engines, such as Mercator [41], limit the maximum number of requests to a single host by one request per second. From Chapter 2, Section 2, we know a significant portion of the requests are HTTP HEAD requests, which impose much less load on web servers and network traffic. For quality concerns, we care about the accuracy of extracting changes and the quality of presenting information. The techniques for extracting web changes and evaluating new information discussed in Chapter 2 are used in the development of our application. Furthermore, as an online application, the service should achieve high availability for continuous service.

3.2 Architecture

Although the service and the techniques provided by WebDNA are quite different from public search engines, the fundamental framework of the application is roughly similar. It contains three major components: an incremental crawler, a data repository and a query engine. In addition to these, there is a Web portal to make it accessible through the internet. We store and manage data in a relational database such that a word indexer is not included in our application. Figure 3.2 shows the architecture of this application in an overall view.

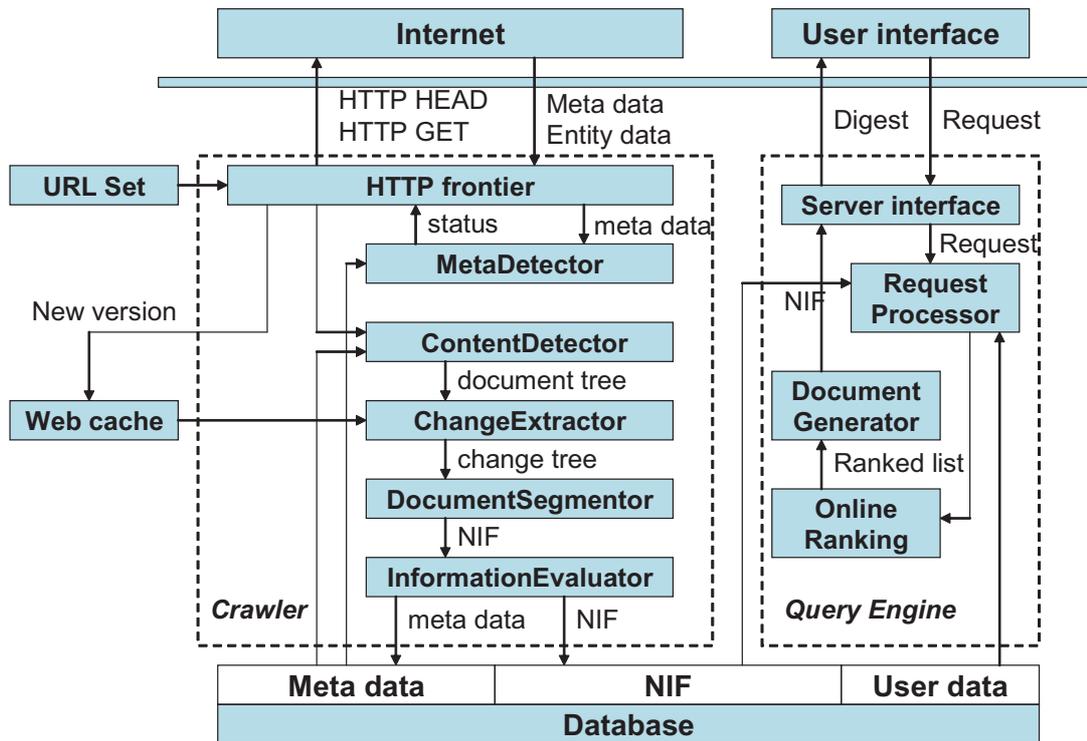


Figure 3.2: The architecture of WebDNA.

The Incremental Crawler starts from a pre-built set of URLs. It repeatedly probes and downloads the web pages from Internet to maintain a fresh database. There are five major kernel components that are integrated into the Incremental Crawler:

- *MetaDetector*: detecting pages that have been changed since last visit using HTTP meta data.
- *ContentDetector*: detecting pages that have been changed since last visit using content entity.

- *ChangeExtractor*: extracting changed content between the downloaded version and cached version.
- *DocumentSegmentor*: dividing changed content into NIFs.
- *InformationEvaluator*: evaluating the static scores of NIFs.

The techniques for building these components have been discussed in Chapter 2.

The crawler manages a URL set to detect or download. For each URL, the crawler first issues an HTTP HEAD request to the web server and passes the meta data to *MetaDetector*. If the *MetaDetector* finds that a page has been updated or if it cannot determine whether the content has been changed, it tells the crawler to download the actual content of the web page. Then the document tree of the current version of web page is constructed and *HT-encoding* is computed. If the *MetaDetector* returns a status of *UNKNOWN*, the *ContentDetector* compares the *HT-encoding* string of the root node of the document tree with the encoding string of the last version in our database. Once *ContentDetector* determines that the page has been changed, the encoded document tree of current version is passed into *ChangeExtractor*. Any web page in the URL set has a cached copy of last version of the web page on local disk. The *ChangeExtractor* gets the old version from local disk, performs tree matching of two different versions and extracts what has been changed in the new version. It returns a *ChangeTree* that contains all changed data. The *DocumentSegmentor* partitions the *ChangeTree* into several content-independent text fragments. Then the *InformationEvaluator* evaluates these text fragments to determine which are valuable information sources, each of which is call a New Information

Fragment (NIF) that will be stored into data repository with related Meta information. In each round of crawling or probing, if any page has been updated, a new cache of the content is stored on the disk and the corresponding updated Meta data is stored in database.

The Query Engine is a standalone server application, which may be used by multiple users simultaneously. It is authorized to read data on relational database but not authorized to write to the database. A script program that is deployed at web server accepts and pre-processes requests from web interface. The script program forwards the requests to Query Engine via sockets. Query Engine maintains a server socket to accept requests and uses a multi-threaded model to distribute work to Request Processors. Each request processor talks with the database to retrieve recent NIFs properly. Each NIF comes with a static ranking score. The online ranker then orders them into a ranked list using evolution ranking. Typically, the multi-threaded application model is very similar to a simple web server.

3.3 Building the initial data

Before starting the crawler and query engine, we need to collect and set up the following data:

- URL set.
- Popularity ranking scores.
- Initial document cache.

- The meta data of initial document cache.

In order to collect all these data, we run a navigational crawler, which starts with a small URL set and crawls the NYU web by following the links on web pages. The navigational crawler maintains a URL queue for downloading and a hash table storing all visited URLs. The crawler picks URLs from the front of the URL queue to download while newly found URLs are added to the end of the queue. This crawling mode corresponds to a breath-first-search of the web. As Najork and Weiner [51] studied, the breath-first-search crawling mode can produce good crawling results in practice. However, a strict breath-first-search crawling mode has the following problem: as many pages contain large number of links pointing to the same server, the frontier of the URL queue can be aggregated by URLs on the same host. This imposes a high load for web servers. To reduce such negative effect, we re-arrange the crawling queue by randomly inserting a newly found URL into the last 1000 URLs of the same depth in the queue. Such re-arrangement only applies to URLs of the same depth during crawling such that the overall crawling mode still preserves breath-first-search order. It is very possible that navigational crawler can jump into a *crawling trap* [41], where the URL queue is piled with mass URLs of a huge sub-collection of the web, e.g. URLs randomly generated by scripts. In order to overcome this problem, we manually monitor an exclusion list of URL prefixes during crawling.

The navigational crawler collected over 600,000 different URLs within the domain of nyu.com. It is interesting to know how much information is retrievable as hypertext documents. Figure 4.2 shows the HTTP response distribution and media type distribution of successfully retrieved URLs. The category

of “successful” HTTP responses includes the HTTP response code of 200, and redirections to URLs that get 200 response code too. The category of “dead-links” includes URLs with 4xx and 5xx HTTP response codes, unknown host, server timeout, etc.

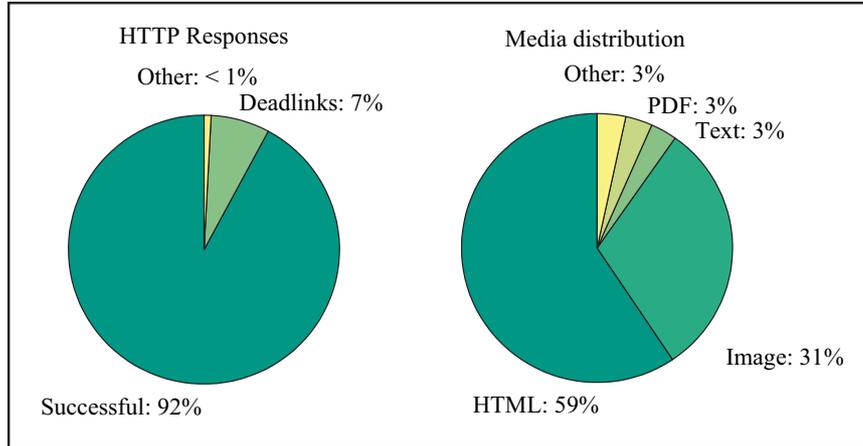


Figure 3.3: HTTP response and media type distribution of URLs in NYU domain.

Because mass junk pages exist on the web and the limit of local disk space, the navigational crawler only collects URLs, link structure and meta data. Then we build the web graph using the link structure and compute the popularity scores using popularity ranking algorithm. URLs with very low popularity scores are excluded from our URL set for WebDNA. URLs pointing to non-html content and many low quality dynamically generated links are removed too. After ranking and filtering, about 200,000 URLs are kept. Then we download the pages in this URL set to build the first local cache. The URLs are spread on 279 different hosts. We group these hosts into 119 groups according to their third-level domain name, e.g. host *skirball.med.nyu.edu* is grouped into domain

med.nyu.edu. Then we sort the domains by the number of pages. Figure 3.4 shows the top domains in NYU web:

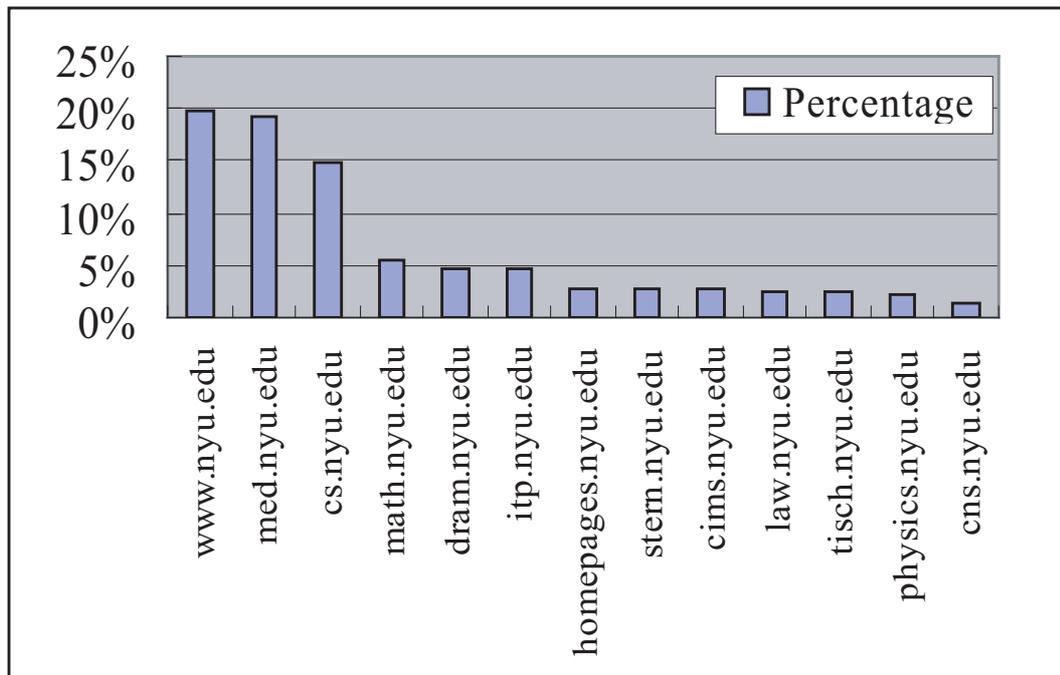


Figure 3.4: Top sub-domains in NYU web.

For data management, all meta data are stored in relational database while the cached web pages are stored on disk. We choose free database software MySQL for our use.

3.4 An event-driven incremental web crawler for WebDNA

Built on LibWWW [4], the crawler uses an event driven architecture to manage tasks and data flow. Figure 3.5 shows the event loop in the crawler. With the

prebuilt URL set, the crawler starts with one HTTP request for a URL in the set to enter into event loop. In the request/response diagram of LibWWW, all non-internet operations on the data getting through or getting from network are performed by *handlers*. Some handlers are registered before or after the network activity during the event loop. The crawler maintains a pool of active requests or connections to the internet. Some handlers are registered during parsing the content. The functionalities of these handlers are shown as follows:

- Net-before handler is called before a request is sent to the internet. It validates the availability of a URL from the result of last visit. If the local data shows that the host of this URL was available but the web page of the URL is not available since last visit, the request is disregarded and will not enter into the network module.
- Net-after handler is called after a request is finished. Once a request is finished and the data is processed, its network after handler checks the size of current pool and starts new requests; if the pool is not full and there are new requests pending, it starts a number of new requests to fill the pool before deleting itself. The net-after handler also determines when a round of crawling is finished by checking the activity of the request pool.
- All handlers processing hypertext data are registered in the network module, including hypertext document creation and deletion callbacks, hypertext document build callback, text data callback, start and end tag callbacks, and hyperlink callback. These handlers parse the content, build the document tree, encode the document tree and process other tasks if necessary. Four kernel components described in Section ??, *ContentDetector*, *ChangeExtractor*, *DocumentSegmentor* and *InformationEvaluator*

are directly called by hypertext document build handler.

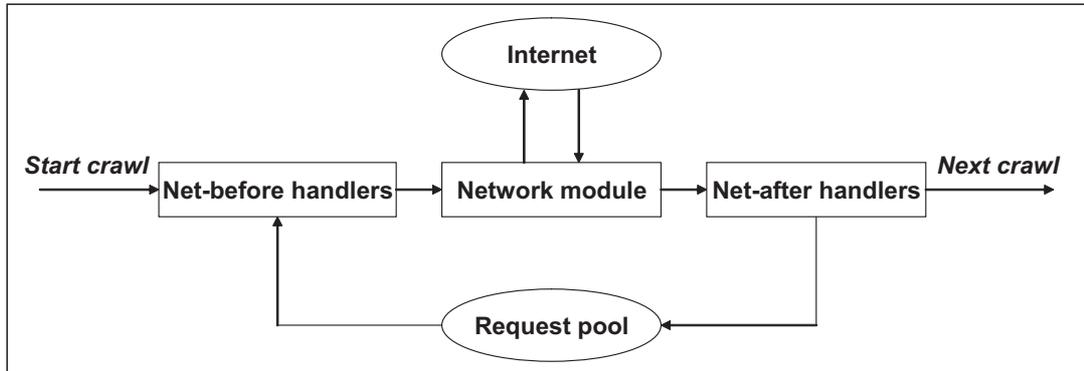


Figure 3.5: The eventloop of incremental crawler.

Because many requests are processed in parallel, it is difficult to determine which will be finished earlier than any other in the event loop. Therefore, in a single-thread model, we encounter two problems: first, if a request blocks for a long time and the timeout operation fails, the request will stay in the pool forever; second, the request finishing the current round of crawling cannot be determined by itself. To solve these problems, we run our crawler in two thread mode. One thread is an event-driven crawler, and the other is a crawling control thread. The active request pool is shared by both threads. Both threads also share a status flag which stores the current crawling state (i.e. crawl-start, crawl-end, etc). The crawl control thread periodically checks the active request pool. If a request is inactive in the pool for a long time, the control thread kills and removes it from the pool. If there is no more new requests pending and no active requests, it revises the status flag to initiate the ending of current round of crawling. The net-after handler of the last request will detect the state flag until it has been changed as crawl-end. Then it terminates all active connec-

tions (the idle persistent HTTP connections without any active requests) and starts the next round of crawl after a period of sleep. A script program is used to schedule crawls.

3.4.1 Determining the crawling rate

How rapidly should our crawler send requests to web hosts? From Figure 3.4, we know that the largest sub-domain is a single host in NYU web, www.nyu.edu. It takes about 1/5 of the whole local web. If the crawler runs too fast, it imposes an significant load on the host and affects its performance in serving other users. In order to find the proper crawling rate, we define the following metric to measure the traffic generated by the crawler to web servers:

Definition 6. Let $\tau(t)$ be the maximum number of requests to a single server sent by the crawler during $[t, t+1)$ and $n(t)$ be the total number of requests sent by the crawler during time $[t, t+1)$, the **loading ratio** to web servers is defined as the mean of $\tau(t)/n(t)$ during the whole crawling process, denoted as ζ .

The *loading ratio* depends on the crawling order of the URL set. We discuss the effectiveness and advantages of two orders here: random order and optimal order.

Random order

We generate a random order of all URLs. Then we simulate the behavior of the crawler by following the same order. Suppose we send N requests per second. We compute the *loading ratio* of different settings of N on our URL set. The result is given in Figure 3.6. The *loading ratio* decreases as N increases.

However, since the largest host serves about 20% of all web pages, the ratio will never be lower than 0.2 no matter how large N is.

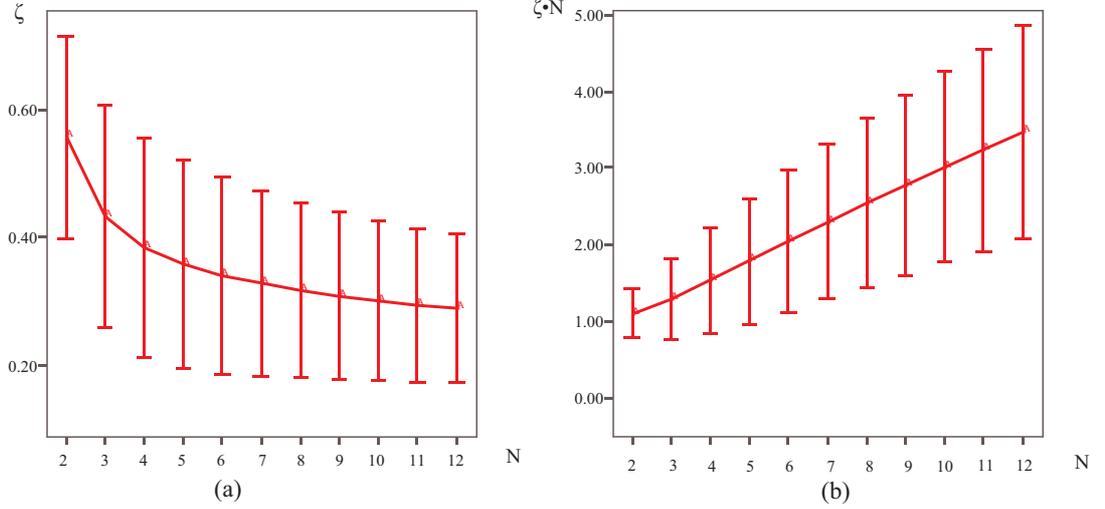


Figure 3.6: The loading ratio and crawling load on web servers. (The error bar is mean +/- standard deviation).

For random order, an appropriate crawling rate is 8 requests/sec. When $N = 8$, the average maximum number of requests that the crawler sends to the same host during each second is 2.52, which is considered very “polite” to web servers. From the standard variation bar, we can estimate the distribution of $\tau(t)$, the maximum number of requests that the crawler sends to the same server during each second. Let $\sigma(\tau)$ be the standard deviation of $\tau(t)$. From the knowledge of statistics, we know that most $\tau(t)$ is within the range $[\text{Mean}(\tau(t)) - 2\sigma(\tau), \text{Mean}(\tau(t)) + 2\sigma(\tau)]$. For example, if the data follows a normal distribution, above 95% of the cases fall into that range. From Figure 3.6(b), we see that for $N = 8$, $\text{Mean}(\tau(t)) + 2\sigma(\tau)$ ¹ is less than 5, which is

¹ $\zeta = \text{Mean}(\frac{\tau(t)}{N}) = \frac{\text{Mean}(\tau(t))}{N}$, therefore $\text{Mean}(\tau(t)) = \zeta \cdot N$.

tolerable. When the crawling rate is 8 requests/sec, the crawler can roughly send 690,000 requests each day, which is enough to maintain the freshness of our web index.

The order of minimum loading ratio

In fact, the order of URLs can be arranged to get the minimum loading ratio, which equals the fraction of pages on the largest host. Consider we have M pages, and the largest host H_0 have ηM URLs, where $0 < \eta < 1$. Then we divide the ordered list into ηM buckets, where each bucket contains $\frac{1}{\eta}$ URLs. Listing 3.1 shows the algorithm for generating the optimal order. Because the number of buckets is greater than or equal to the number of URLs on any host, each bucket contains at most one URL from a single host. Furthermore, any $\frac{1}{\eta}$ successive URLs contain at most one URL from a single host too. Hence the average loading ratio for this order is η . For NYU web site, $\eta = 0.2$. Consider we can send 2.5 requests per second to a single server, the crawling rate for this order is 12.5 per second, which is larger than the crawling rate for random order.

Listing 3.1: The algorithm for generating the optimal order

```
1 Given :
2   M — number of URLs
3   K — the size of each bucket
4 Output :
5   an ordered list L
6
7 Init :
8   Generate an ordered list L0 for all URLs grouped by host
```

```

9     i , offset = 0
10
11  For each URL u in the list L0 {
12     L[i+offset] = u;
13     i += K;
14     if ( i >= M ) {
15         i = 0;
16         offset++;
17     }
18 }

```

Although optimal order gives better crawling capacity than random order, it is not easily extensible. Each time when we add new URLs into the set, we have to recompute the order. For random order, adding new URLs is much simpler. We can randomly insert new URLs into the set without any restriction. Since the crawling rate based on random order is fairly enough for our purpose, we choose random order for crawling.

We keep a timer to separate successive requests to adjust the crawling rate to be close to 8 requests per second. Currently, it issues about 6.3 HTTP HEAD requests per second and 1.7 HTTP GET requests per second on average. The average downloading size, including both meta data and content data, is about 20 KB per second, which is a tolerable internet traffic on the local web. During Mar. 2005 to Jan. 2006, we kept the crawler running almost all the time except for several times of redeployments of code improvement. A few interruptions were caused by network problems.

3.5 Retrieving web changes from database

The query engine of WebDNA is a server which process all kinds of user queries. Each request is processed by the following steps: identifying the type of requests, correcting user data if necessary, constructing the corresponding SQL query, retrieving NIFs from database and formatting results into a web page. We know the design of data schema has great impact on data retrieval. According to the types of user requests, the data schema for NIFs contains multiple indexes as follows:

- Index on URL, which is used for retrieving change history of a single web page.
- Index on host name, which is used for retrieving changes on a single host.
- Index on the last modified time, which is used by multiple queries with time limit.
- Index on the static ranking score, which can improve the performance of ordering.
- Full-text index on the title of web pages and content of web changes, which is used by full-text search.

Our ranking algorithm consists both static ranking and evolution ranking. The static ranking score can be computed at crawling time and stored in the database. However, the evolution ranking uses the query time, which cannot be stored as a column in the data table. Therefore, the “order by” clause in SQL query must contain a formular which computes the ranking score using both static score and query time. We compare the query performance using and not

using index on static ranking score. Experiments shows that using index on static score does improve the performance by 30% on average.

3.5.1 Full-text keyword search

The full-text keyword search on web changes is built based on both the full-text search provided by database software and our ranking algorithms. MySQL provides a full-text search which orders results using text relevance. However, the text relevance ranking is not good for presenting web information. We combine the relevance score and our ranking framework to present the most important and relevant web changes to user queries. Mathematically, there are numerous ways to combine the relevance score and query independent score. We compare two fundamental methods, “product” method and “sum” method, and we give the descriptive reasons for our choice of “product” method rather than “sum” method. Let S_{query} be the relevance score to query and $S_{non-query}$ be the query independent score given in Eq. 2.10. The final score of “sum” method is $S_{query} + S_{non-query}$ and that of “product” method is $S_{query} \times S_{non-query}$. Our purpose of ranking is to return both highly relevant and important changes on the top on the returned list. Consider the following examples:

- Record 1: $S_{query} = 3.0$, $S_{non-query} = 3.0$.
- Record 2: $S_{query} = 0.5$, $S_{non-query} = 6.0$.
- Record 3: $S_{query} = 6.0$, $S_{non-query} = 0.5$.

Using “sum” method, both Record 2 and 3 are ranked higher than Record 1. But using “product” method, Record 1 is ranked higher than Record 2 or 3. Which ranking is better? First, if we compare Record 1 and 2, we see that Record 2

has low relevance even the page is an important page on the web. People may not find good information of they interest from it. Second, we compare Record 1 and Record 3 and find Record 3 has much higher relevance, but its query independent importance is low. It can be un-popular pages that few people reference to it. Even Record 3 has higher relevance, but if few people like it and reference to it, it is highly possible that the page is of little significance to user's interest or even a keyword spam. Therefore Record 1 should be ranked higher than both Record 2 and 3, and "product" method is better.

The following is the MySQL query that we use for full-text search:

```
SELECT * FROM
  (SELECT [columns],MATCH (title,content) AGAINST ([keywords])
    AS relevance
  FROM niftable
  WHERE MATCH (title,content) AGAINST (<keywords>)
  HAVING relevance > [threshold]
  ORDER BY relevance DESC
  LIMIT [threshold of number of records]) AS temptable
ORDER BY
  relevance*staticscore
  *exp(-alpha*(Current time - last modified time)) DESC
```

3.6 Developing tools

WebDNA is implemented and deployed on a single machine. Currently it is running on a PIII machine with 256MB memory and T1 Internet connection. The crawler, the integrated components for processing web data and the query

engine of WebDNA is written completely in C++. Several offline tools and analyzers are written in Java. In order to reduce the work of system level coding (i.e. implementing Internet protocols from bottom up) and reduce potential programming bugs as much as possible, we build our application on two popular and well tested library tools: LibWWW and Xerces C++.

LibWWW, the W3C protocol library, is the implementation of Internet protocols by W3C.org. As it states, “LibWWW is a highly modular, general-purpose client side Web API written in C for UNIX and Windows (Win32). It’s well suited for both small and large applications, like browser/editors, robots, batch tools, etc.”. The design of LibWWW adopts three-layer software development architecture. The layer in the bottom is a large set of generic utility modules such as container classes, string utilities, network utilities etc. They have the important function of separating the upper layer code from platform specific implementations using a large set of macros that makes the Library more portable. The modules are used throughout the Library itself and some of them have be utilized in our application development.

Xerces is the next generation of high performance, fully compliant, and widely used XML parsers in Apache XML Project. Xerces-C++ is a validating XML parser written in a portable subset of C++ in the Apache Xerces family. Xerces-C++ is faithful to the XML 1.0 recommendation and many associated standards. In particular, Xerces C++ conforms to the following standards: XML 1.0 (Second Edition), DOM Level 1 Specification, DOM Level 2 Core Specification, etc. Xerces has an implementation of DOM structure, which we use to build the document tree view of HTML documents.

Chapter 4

Modeling Web Changes and Analysis

4.1 Introduction

There are a large number of web pages updated every day. However, the change frequencies of web pages varies a lot. Some pages are updated several times per day while some others remain unchanged for months. Accurate estimation of web change frequencies is very important in the following problems: scheduling crawling resources for incremental web crawlers and web catching. Building effective models of web changes can help estimate change frequencies or even predict web changes.

Currently the problem of synchronizing a web index is treated as the problem of synchronizing a database. Each web document is considered as a data element in the database. If the size of the database is considered relatively static, then each element is detected for a fair long period to estimate its average change frequency; and by given limited resource and the knowledge of

change frequencies, optimal synchronizing schedule can be obtained to achieve maximum overall freshness of the database. Cho and Garcia-Molina [24, 25] discussed the methods for crawler scheduling based on this approach. There are some challenges in this approach. First, accurate estimation of the average change frequency requires an examination of a long change history of web documents. When the detecting history is short, the estimation of change frequency has big deviation from the actual one, as we will show in Section 4.3. Second, not like a database, web documents indexed by search engines have very diverse quality. Such diversity has big impact in retrieval. We know that search engines present results as a ranked list. Only those in the top positions are retrieved frequently. Such retrieval pattern is very different from relational database. In evaluating the freshness of web index, we should take into account the retrieval frequencies of web pages.

In this chapter, we focus our studies on how to improve the accuracy of measuring the change frequencies of web pages. First, we model web changes using the characteristics of web documents and find PageRank value [55] is a good predictor of change frequency. Second, by combining the predicted frequency and the one derived from partial detecting history, we obtain an improved estimator.

The following of this chapter is organized as follows: in Section 4.2, we introduce necessary background knowledge that we will use in this chapter; in Section 4.3, we first review the current studies of modeling web changes, then discuss the methods and solutions of our approach; in Section 4.4, we present our experimental results using the data we collected from the New York University web site.

4.2 Background

4.2.1 Change frequency and change history

The change frequency is a metric measuring how often web pages are modified. It can be used to improve the scheduling of web crawlers [24] and web caching [12]. Brewington and Cybenko [13, 14] and Cho and Garcia-Molina [25] both identified that the changes of web pages follow Poisson Process. A Poisson process is often used to model a sequence of random events that happen independently with a fixed rate over time. If a web page follows a Poisson Process with change rate λ , then the probability density function of the time to the occurrence of the next change is

$$f(t) = \begin{cases} \lambda e^{-\lambda t} & \text{for } t > 0 \\ 0 & \text{for } t \leq 0 \end{cases}$$

A change history of a web page is a set of change events made to the web page. A partial change history can be obtained by detecting the web pages regularly. However, detection based change history can be inaccurate. First, there may be multiple changes between two successive detections. Second, detections are often made in fixed period of time and is incomplete. The accuracy of the detection based change history can be improved by increasing the detection rate.

4.3 Modeling web changes

Two kinds of data sources can be used to model the changes of web pages: the change history and the characteristics of web pages, e.g. length, location, importance, etc. The methods used to model web changes using change history

alone are discussed by Cho and Garcia-Molina [25]. In this section, we first review the frequency estimators presented by Cho and Garcia-Molina. Then we analyze these estimators and find these estimators can be inaccurate in certain situations. In Section 4.3.3, we describe our approach of identifying the characteristics that can be used to predict change frequencies. Section 4.3.4 presents an improved frequency estimator which combines the predictors and change history.

4.3.1 Review of frequency estimator based on detecting history

Cho and Garcia-Molina [25] studied how to estimate the change frequencies of web pages using an incomplete detection history. Here we briefly introduce the estimators they presented. Suppose a web page is detected n times at frequency of f times per time unit, and among them, X detections find that the page is modified. Consider the detection engine runs under a fixed amount of resources (CPU, network, etc) and the size of the collection of pages to detect is relatively static, then the detection frequency f can be considered as a constant and the time interval, $I = 1/f$, for every successive two detections on a single page is a constant too. Denote $r = \lambda/f$. A naive estimator of r is given by

$$\hat{r} = \frac{X}{n}$$

where \hat{r} is the estimated value. As the number of detections n increases, the estimation gets more and more accurate. However, Cho and Garcia-Molina [25] showed this estimator is not good enough and proposed an improved estimator

which is given as follows:

$$\hat{r} = -\log\left(\frac{n - X + 0.5}{n + 0.5}\right) \quad (4.1)$$

Cho and Garcia-Molina [25] proved that this estimator is consistent eventually by showing $\lim_{n \rightarrow \infty} E[\hat{r}] = r$ and $\lim_{n \rightarrow \infty} V[\hat{r}] = 0$, where $E[\hat{r}]$ and $V[\hat{r}]$ are the mean and standard variation of \hat{r} respectively.

4.3.2 The disadvantage of detection based frequency estimator

Web evolution studies [54, 36] show that the change frequencies of web pages varies significantly. A large amount of web pages follows a change pattern with very small frequencies. Here we show that the change frequency estimator based on an incomplete detection history is not accurate when r is small, even though the standard error converges to 0 eventually.

In Poisson Process model, the probability that a page changes during a time interval I is e^{-r} . Then we get the probability that X equals i as follows

$$Pr(X = i) = \binom{N}{m} (1 - e^{-r})^i (e^{-r})^{N-i}$$

Then

$$E[\hat{r}] = -\sum_{i=0}^n \log\left(\frac{n - i + 0.5}{n + 0.5}\right) \binom{n}{m} (1 - e^{-r})^i (e^{-r})^{n-i}$$

The standard error is given by

$$V[\hat{r}] = E[\hat{r}^2] - E^2[\hat{r}]$$

The above formulation provides a way of how to compute $V[\hat{r}]$ theoretically.

The standard error $V[\hat{r}]$ measures the absolute deviation of the estimated value

from the actual value, which does not take into account the magnitude of actual change frequency. Alternatively, we use the ratio of standard error, $V[\hat{r}]/r$, to measure the deviation. Figure 4.1 shows the theoretical values of the ratio of standard error for estimator given in Eq. 4.1. Given a fix number of detections, the error ratio increases dramatically as change frequency decreases. It is very difficult to make accurate estimation for pages with small change frequencies.

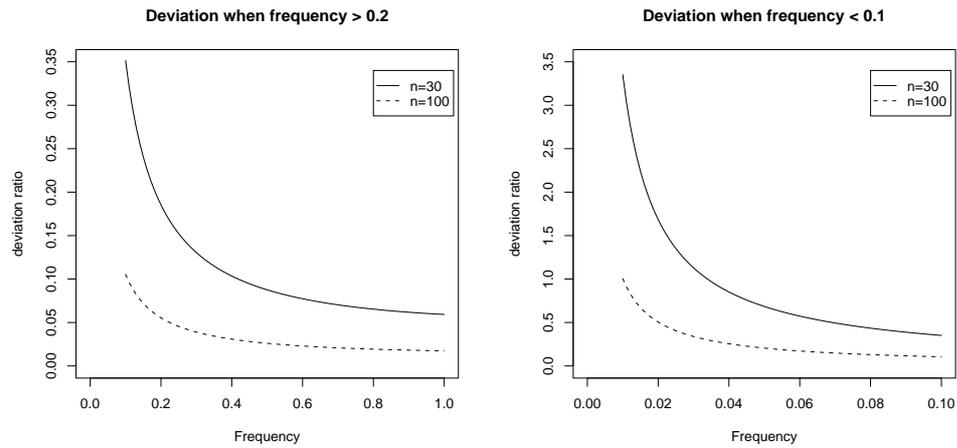


Figure 4.1: The ratio of standard error of change frequency estimator.

4.3.3 Predicting the change frequency of web pages

We learn that estimating change frequency of web pages using detection history is not always a reliable method. Here we describe our “survival analysis” modeling approach for identifying the features of web pages that can be used to predict change frequencies.

Survival analysis and Cox Proportional Hazards Regression model

Survival analysis encompasses a wide variety of statistical methods for analyzing the timing of events, also known as *time to event analysis*. These methods were initially used to predict the time of survival for patients under different treatment, where the prototypical event is *death*. But survival analysis is also appropriate for many other kinds of events, such as job changes, marriage, birth of children and so forth. For our purpose, an event is a modification of a web page and the *survival time* is the lifetime of a version of the web page. Survival times can be modeled through a *survival function* $S(t)$, which is the probability that the survival time of an object is greater than or equal to t . The distribution of $S(t)$ is also described in terms of a *hazard function* $h(t)$, which is the “ratio of failure” at time t . The hazard function is defined as the following:

$$h(t) = -\frac{\frac{dS(t)}{dt}}{S(t)}$$

For modeling web changes under Poisson Process, we have

$$S(t) = \int_t^{\infty} \lambda e^{-\lambda \hat{t}} d\hat{t} = e^{-\lambda t}$$

The hazard function $h(t)$ for a single web page under this model is a constant, which is the rate of change (λ). Modeling on a single web page is studied by Cho and Garcia-Molina [25] as we reviewed in Section 4.3.1. However, what parameters that can affect the value of λ are unknown. We need to find the predictor variables that influence the survival time.

The Cox Proportional Hazards Regression method in survival analysis is widely used in statistics for discovering important variables that influence survival time. Let $x(p)$ be a predictor variable of data element p , under the Cox

model, the hazard function $h(p, t)$ can be expressed as:

$$h(p, t) = e^{\beta x(p)} h_0(t) \Rightarrow \ln h(p, t) = \ln h_0(t) + \beta x(p) \quad (4.2)$$

where $h_0(t)$ is the baseline hazard function, common for all elements in the population. The Cox model can be generalized for n predictor variables: $\ln h(p, t) = \ln h_0(t) + \sum_{i=1}^n \beta_i x_i$. An important feature of Cox model is that it can effectively exploit incomplete or “censored” data, from cases that “survived” the whole studying period. For modeling the changes of web pages, a lot of web pages do not change for a long time period and it is not practical to obtain sufficient change history for all of web pages. These pages constitute a significant portion of the web and generate a large amount of censored data for study. The censored data contains partial information but excluding them can seriously affect the results. Another important feature of Cox model is that it does not assume any shape of the base line function.

Using COX regression to model the changes of web pages

In modeling web changes, we assume each single page follows a Poisson Process such that the survival time follows an exponential distribution. The baseline hazard function is independent of time under this model, we simply denote $h_0(t)$ as λ_0 . Now we consider what features can be used as parameters. The features we use are:

- The logarithm of PageRank score.
- The content length of web pages.
- The length of the newly added content in each change.

The PageRank score is an evaluation of the popularity of web pages. This feature is selected to examine whether popular pages gain more updates. We know PageRank score is less than 1 and many web pages have small PageRank value. We add a constant over the logarithm of PageRank such that the feature value is greater than 0. We call it *normalized logarithm of PageRank*. We find only the PageRank score is a good predictor of the survival time of web pages experimentally. The average of the coefficient of the normalized logarithm of PageRank for multiple studying periods varying from 2 month to 8 months is 0.36 given the following settings: the logarithm of PageRank score is normalized to be in the range of $[0, 10]$. Section 4.4.2 discusses our experiments in detail. This unveils the following trend how web authors choose to update web pages: popular pages are more likely to be better maintained to be up-to-date. Given the PageRank score $PR(p)$ of a web page p , the predicted change frequency is:

$$\lambda(p) = e^{\beta(\ln(PR(p))+C)} \cdot \lambda_0 \quad (4.3)$$

4.3.4 An improved change frequency estimator

Although Cox model uses a change history to identify the predictors of survival time, the experiments in Section 4.4 will show that the coefficients of the predictors is stable regardless of the length of the change history. This provides a way of estimating change frequency without knowing any change history data of web pages. For the estimator derived from change history, we know that the estimator gets more accurate when the detection history gets more complete. Now we consider how to combine both to provide a more accurate estimator. The idea is that in the early stage of detection, we tend to use the predicted estimator, however, as the detection period gets longer, the estimator transits

from predicted estimator to change history based estimator.

Denote the predicted estimator as $\lambda_1(p)$, which is defined as Eq. 4.3. Denote $\lambda_2(p)$ as the change history based estimator of change frequency for page p . From Eq. 4.1, we have $\lambda_2(p) = -f \cdot \log(\frac{n-X+0.5}{n+0.5})$. Then improved estimator is given as follows:

$$\lambda_{imp}(p) = e^{-\alpha t} \lambda_1(p) + (1 - e^{-\alpha t}) \lambda_2(p) \quad (4.4)$$

where α is a configuration parameter determining how fast the transition between different estimators. When $t = 0$, $\lambda_{imp}(p) = \lambda_1(p)$. When $t = \infty$, it goes $\lambda_{imp}(p) = \lambda_2(p)$. Therefore, $E[\lambda_{imp}]$ eventually converges to the actual change frequency and $V[\lambda_{imp}]$ converges to 0. The effectiveness of the improved estimator based on incomplete history is discussed in Section 4.4.

4.4 Experiments

The data we use for our analysis is collected from the web of New York University. We ran a crawler and collected about 240,000 web pages. Then we constructed the link structure of these pages and computed the PageRank scores. We select the top ranked 60,000 web pages for detection. We detect each page once every day and obtained a complete change history from March 2005 to November 2005. Although the NYU web site is much smaller than the global web, it is structurally similar. The ratio between the number of hyperlinks and URLs is 7.4, which is in agreement with previous work [17]. The study of the in-link and out-link distributions shows they obeys power law with power coefficients of 1.94 and 2.24. Both of them agree well with previous work studied by Broder et al. [17] and Barabasi and Albert [11]. The average size of the web documents is 10.5K bytes similar to the results given by Fretterly et al.

[36]. In order to study how popularity metric affects the change frequencies, we first analyze through a simple regress method on “uncensored” data only, then discuss the effectiveness of Cox model using both “censored” and “uncensored” data.

4.4.1 Simple regressions on uncensored data

Given the PageRank scores of the 60,000 pages in our data set, we choose a constant C such that the values of the normalized logarithm of PageRank , $\ln PR + C$, are mapped into the range $[0, 10]$. We notice that the number of cases that $\ln PR + C$ is greater than or equal to 6 is small. Thus all pages are divided into 7 groups: group i , $i < 7$, contains pages that $i - 1 \leq \ln PR + C < i$; group $i = 7$ contains pages that $\ln PR + C \geq 6$. For each group i we choose a representative value of the normalized logarithm of PageRank, denoted as $\phi(PR)$. For each group $i < 7$, $\phi_i(PR) = i + 0.5$. For group 7, we choose $\phi_i(PR) = 6.5$. Figure 4.2 (a) shows the size of each group. The plot shows the logarithm of the number of pages in each group is a linear function of the representative value of normalized logarithm of PageRank scores. Consider that $\phi_i(PR)$ is the logarithm of PageRank scores plus a constant, such relationship actually shows that the distribution of PageRank values follows a *power-law distribution*, where the number of pages in each group is proportional to the power of PageRank score: $1/(PR)^\alpha$. The coefficient α given by the regression in Figure 4.2 (a) is 1.68. Power-law distribution has been found for the distribution of the degrees of in-link and our-link of web pages [17]. The degree of in-link is one of the metrics measuring the popularity of web page. Here we showed another popularity metric, PageRank, also follows a power-law distribution. For

each group, Figure 4.2 (b) shows the percentage of web pages that have changed during the period of our study. It does exhibit the behavior that pages with higher PageRank scores are more likely to be modified.

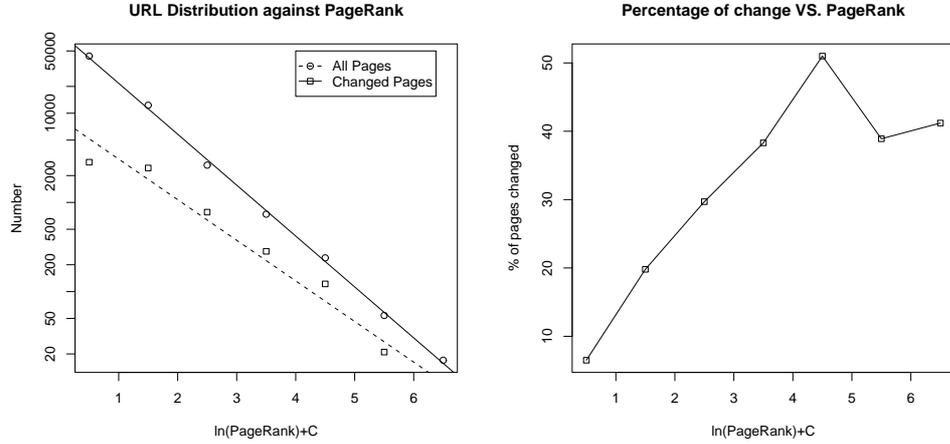


Figure 4.2: The distribution of URLs.

In order to find the statistical trend of the uncensored change data, we did an exponential regression on the average change frequencies over the 7 groups, which follows $\lambda_{\text{avg}}(i) \sim e^{\beta \cdot \phi_i(PR)}$ for each group i . The change frequencies are computed based the entire change history of the period of our study using the method given in Eq. 4.1. Figure 4.3 shows the results of simple exponential regression. The result of the regression gives a coefficient $\beta = 0.390$.

4.4.2 Cox regression on both uncensored and censored data

We choose statistical software R-Project [5] to do Cox regression. Our eight month change history data contains 85,053 uncensored data items and 60,000

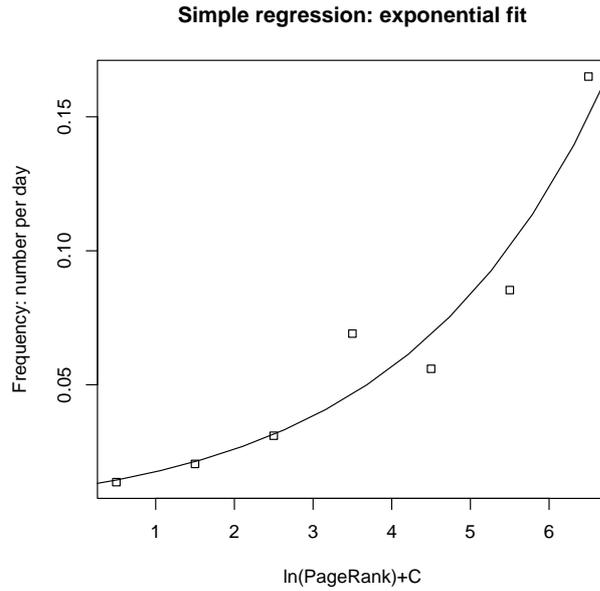


Figure 4.3: Simple exponential regression.

censored data items (including the pages that do not change during the time period of study as well as the pages that changed but censored at the end of the period of study). Since computing cost of Cox regression is high, it is not practical to do a direct regression on all the data set. We use sampling method to build a representative but much smaller data set for Cox regression. From the distribution of PageRank scores (Figure 4.2 (a)), we know that a large number of web pages have small PageRank values. Universal sampling will be greatly biased by the pages with small PageRank values. In order to build a evenly distributed sample set for the logarithm of all PageRank values, we sampled web pages of different PageRank scores with different probabilities. Let $f(\ln PR)$ be the density function of the logarithm of PageRank values and $\text{Prob}[PR]$ be the probability of picking a page with PageRank score PR into

the sample set. Our sampling method is (refer to Appendix E.1 for details):

$$\text{Prob}[PR] \propto \frac{1}{f(\ln PR)} \quad (4.5)$$

In computing the sampling probabilities, we use the following to estimate the value of $f(\ln PR)$.

$$\frac{\text{number of pages in } [\ln PR, \ln PR + \Delta \ln PR]}{\Delta \ln PR}$$

In doing Cox regression, we use incomplete histories of 1 month up to 8 months. For each history data set, we build 5 sample sets, where each sample set contains 500 sample data items. Then we do Cox regression using the PageRank feature for each of the sample sets and compute the average of the coefficients for each history. The results are shown in Figure 4.4. We find other than the history of one month data, the coefficients for the PageRank feature agrees well for different history profiles. The average value from 2 months to 8 months is 0.360. Compared to the results in Section 4.2, we find the coefficient given by Cox regression is similar but smaller.

4.4.3 The effectiveness of improved estimator

We first compute the standard error ratios on our data set using the estimator presented in Eq. 4.1 and observe how they change when the number of detections increases. Figure 4.5 (a) shows the actual deviations and the theoretical deviations. The computation of the actual standard error ratios requires the knowledge of the actual change frequencies. It can only be obtained when the change history is complete. In our analysis, we use the estimated change frequency at the end of our studying period as the actual change frequency in error analysis. We see that the error ratio gets smaller when the number of detections

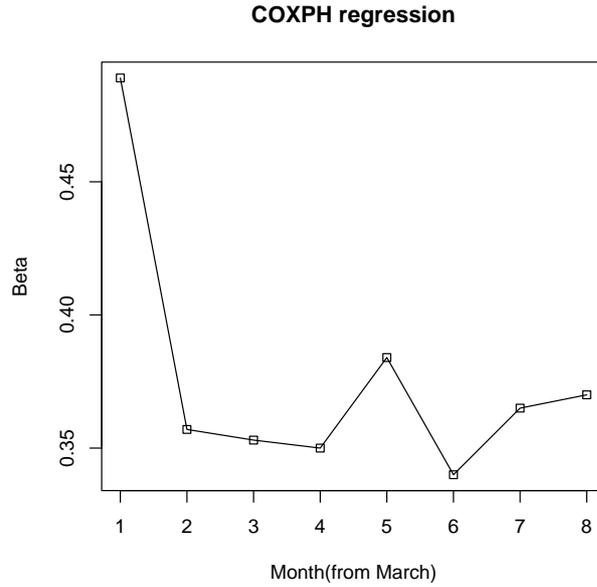


Figure 4.4: Cox regression on different history data sets.

increases. In an overall view, the actual error ratios do agree with the theoretical ones. Figure 4.5 (b) shows the error ratios for different PageRank groups. The trend is similar when number of detections increases. And it is shown the PageRank value does not affect the error much.

In order to evaluate the effectiveness of the improved estimator given by Eq. 4.4, we compute the standard error ratios using the improved estimators. One problem is how to choose α in Eq. 4.4. We compare two configurations here: $\alpha = 0.03$ and $\alpha = 0.01$. With configuration $\alpha = 0.03$, the coefficient $e^{-\alpha t}$ decreases to 0.5 after 23.1 days and decreases to 0.25 after 46.2 days. With configuration $\alpha = 0.01$, the coefficient $e^{-\alpha t}$ decreases to 0.5 after 69.2 days and decreases to 0.25 after 138.6 days. Figure 4.6 shows the results. We found different configurations of α affect the result a lot. The configuration of $\alpha = 0.03$

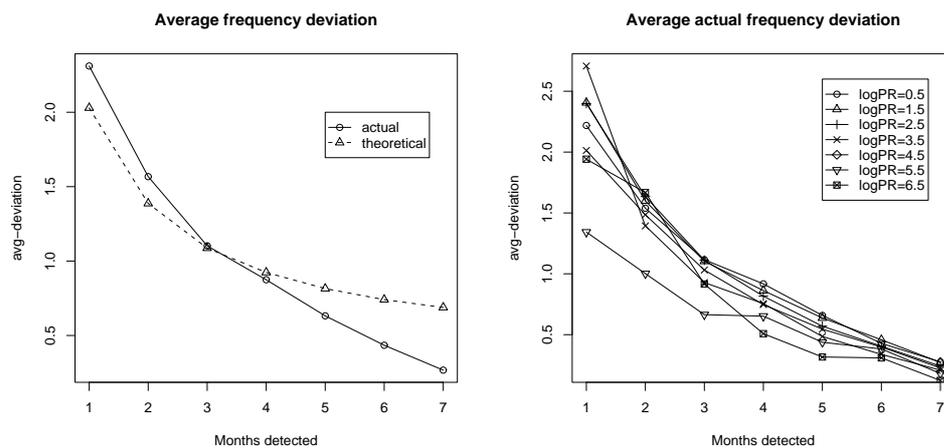


Figure 4.5: The error ratios of change history based estimator.

is superior to the configuration of $\alpha = 0.01$. For $\alpha = 0.03$, when the change history is short, we observe a good improvement. For one month detection history, the improvement is 27.3%. Such improvement lasts for 4 months data. After that, the estimator transits to change history based estimator well. For configuration $\alpha = 0.01$, it does give an improvement during the first months, but after that it is not as good as the change history based estimator alone.

4.5 Conclusion

We presented a study of modeling web changes using survival analysis. We successfully discovered that the PageRank score is a good predictor of change frequencies of web pages. Such predictor can be combined with change history date of web pages to improve the effectiveness of the estimator of change frequencies. As we learned, popular pages are more likely to be updated by web authors. A further thought is that whether updates on popular pages are of

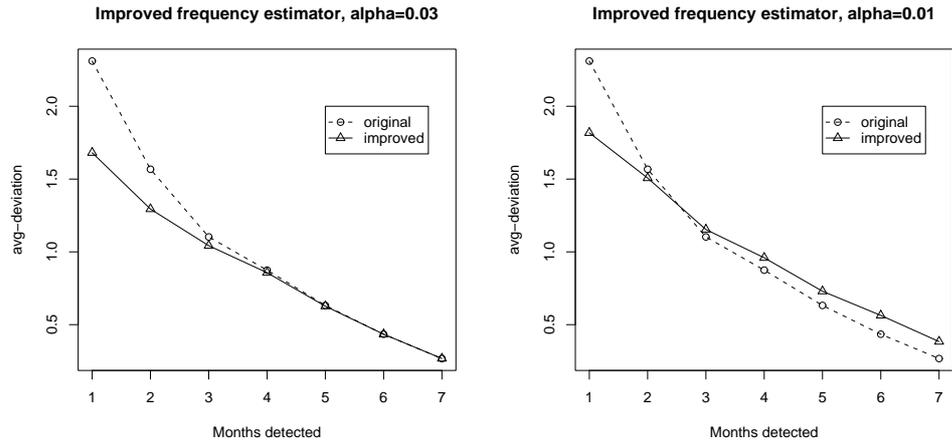


Figure 4.6: The error ratios of the improved estimator.

high quality for retrieval. Cho et al. [27] proposed the concept of page quality, which is closely related to the popularity metrics of web pages. In our prospect, if such quality metric can be used to evaluates the updates of web pages, then people may be able to develop solutions to improve the quality, as well as freshness, of web index for retrieval. We leave it as a good future direction that how different metrics (change frequency, quality, popularity, etc.) can be used in a unified framework for web index synchronization problem.

Chapter 5

Related work

Our work is closely related to many works within the fields of web search and databases. The problems that we have encountered in this thesis come from different subareas, such as web crawling, hypertext information extraction, etc. In this chapter, we first introduce several survey papers discussing the general framework of search engine architecture and metrics of measuring web characteristics. Then we review four categories of related works: web evolution studies, web crawling and synchronization, information extraction of hypertext documents and web ranking.

5.1 Searching the Web

The general framework of search engine architecture and the issues for each component is discussed by Arasu et al. [9] comprehensively. Figure 5.1 is the architecture of a general search engine. It consists of five major components: crawler, indexer, data repository, ranking and query engine. The crawler downloads web pages from the Internet periodically. Giving the crawler certain

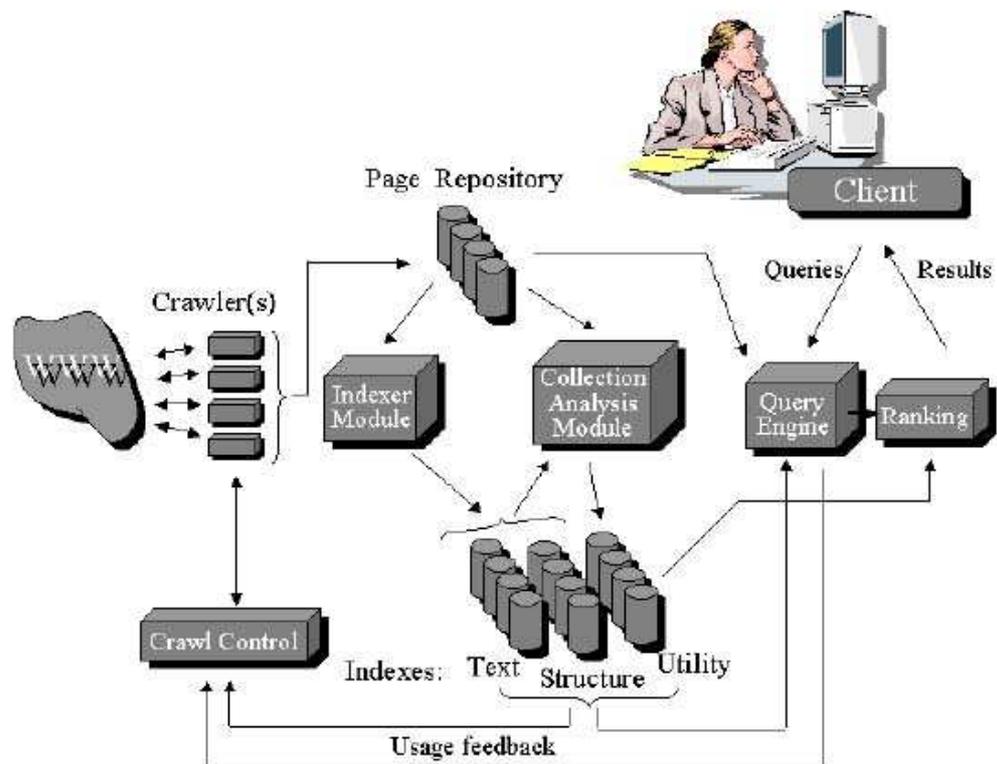


Figure 5.1: General search engine architecture (from [9])

amount of CPU and network bandwidth, the goal is to find and download high quality web pages as much as possible. Therefore, page selection and resource allocation are two critical problems in building a high quality, high performance web crawlers. The indexer extracts all the words from each page and builds an inverted index. The query engine receives search requests from users and return results as a ranked list. The rank of a page is determined by a combination of a query-independent value, such as PageRank, and a measure of the relevance of page to the query. The local copies of web documents as well as the index are stored in the page repository. The Google File System [37] is an

industrial implementation of search engine data storage where the performance of applications is boosted by mass replication of data.

Dhyani et al. [31] reviewed a lot of metrics for measuring the characteristics of the Web structure and content, such as the relevance of web documents to queries, the importance of web pages, and the similarity between web documents.

The relevance of a document D to query Q is computed as follows. Let N be the number of documents in the collection and let M be the number of distinct words. For each word i and document j , let TF_{ij} (term frequency) be the number of occurrences of i in j ; let N_i be the number of documents containing i ; and let $IDF_i = -\log(N_i/N)$ (inverse document frequency). Let $w_{ij} = TF_{ij}IDF_i$. We model each document as a vector $(w_{1j}, w_{2j}, \dots, w_{Nj})$; and we consider query to be a very short document. Then the relevance of document D to query Q is taken to be the cosine of angle between these two vectors:

$$R_{DQ} = \frac{\sum_k w_{kD}w_{kQ}}{\sqrt{\sum_a w_{aD}^2 \sum_b w_{bQ}^2}}$$

This algorithm is called TFIDF algorithm [59], which is widely used in information retrieval.

The importance of web pages can be computed from the web graph structure of the web, where each page corresponds to a node and each link linking two pages corresponds to an edge. The best known of these are the PageRank algorithm [55], used in Google; and the HITS algorithm [43], proposed by Kleinberg. They can be viewed as computing the steady-state distribution of various Markov processes over the Web graph. PageRank algorithm is similar to the behavior of a random surfer on the web graph: at each page, it randomly picks a link on the current page and jump to the target page of the link, or, it jumps to

any random page within the web graph with certain probability. The PageRank scores are the stationary distribution of the stochastic process corresponding to the random surf. HITS algorithm is based on relationship between *authorities* - pages that contain a lot of information about a topic, and *hubs* - pages that link to many related authorities. The importance of an authority is measured by the importance of the hubs pointing to it, and the importance of a hub is measured by the importance of authorities that it points to. Content-based similarity are usually measured by shingle method as we mentioned before. In addition to keyword search of web documents, search engines usually maintain a set of content analyzers of web documents that perform many tasks: information extraction, classification, etc. As a subarea of web data mining [44], information extraction in hypertext media tends to discover information on hypertext documents for a particular purpose. Hypertext documents are semi-structured by makeups and are much less organized than collections of text articles. Consequently, the methods and techniques used in web information extraction is very diverse. Information extraction is very useful in developing numerous web applications, such as product search and people search.

5.2 Web evolution studies

The field of web evolution studies is divided into two categories: web content evolution and web graph evolution. Web content evolution studies the content update frequency and the lifetime of web pages. The two works most relevant to this thesis are those of Ntoulas et al. [54] and Fretterly et al. [36], which we have discussed in Section 1.2. Before that, Cho and Garcia-Molina [22] performed an experiment conducted on more than half million web pages over

4 months in order to measure how web pages evolve over time. They showed the following results: 15% of the pages have a change interval longer than a day and shorter than a week; more than 20% of pages had changed daily; more than 40% of pages in .com domain change every day; more than 70% of the pages had change intervals more than a month. One of the focuses of web evolution studies is to measure the change frequencies of web pages. Previous work of studying the changes of web pages focuses on the distribution of change rate and the modeling based on the change history. The most related work is the study of how to estimate change frequencies by Cho and Garcia-Molina [25]. They modeled web changes using Poisson Process and presented a couple of change frequency estimators. As we discussed in Chapter 4, their estimators are not accurate enough when the change history is short and the value of change frequency is small. Our study discover that the popularity metric can be used to predict change frequencies other than change history. It can improve Cho's estimator, especially in the situation where Cho's estimator can fail. Based on the knowledge of change frequencies, Cho et al. [24] discussed the methods of how to schedule web crawlers to maximize the freshness of web index. A similar approach for scheduling the updates of web databases is covered in the study of Ipeirotis et al. [42]. Brewington and Cybenko [13, 14] use last-modified dates to estimate the distribution of change frequencies over a set of pages, but not the change frequency of an individual page.

Survival analysis has been used to model changes of web databases. Ipeirotis et al. [42] used survival analysis to model the content changes in text databases. Our method for finding the predictors of changes is very similar to what they use. However, the events for their study is the change of a single web database rather than a single web page. The survival function of a web database is more

complicated than a single web page, which follows a *Weibull distribution*. The feature set of web databases differs from web pages too.

Web graph evolution studies the distribution and growth of links in the web graph. Broder et al. [17] studied the graph structure of the web and showed the distribution of the degrees of in-link and out-link of web pages follows power-law. The power coefficients are 2.1 for in-link distribution and 2.3 for out-link distribution. Kumar et al. [45] showed that the web graph structure can be used to infer web communities. The growth model of web graph is shown to be similar to many other networks, e.g. social networks. When a new link is added, it has preference to point to popular pages. Such growth model is often referred as *random network*, which leads to a “rich-gets-richer” phenomenon. Barabasi and Albert [11, 8] studied the properties of random networks and showed a power-law distribution analytically.

Web evolution study is motivated by the maintenance of fresh web index [23], web incremental crawlers [22, 28] and web content caching [12]. Another observation of web evolution is that, as people use search engines heavier and heavier, the index of search engines will get an important impact on page popularity. Those indexed by search engines with high rank will gain more popularity because people will have better chance to visit them if they use search engines. This topic is recently covered by Cho and Roy [27].

5.3 Web crawling and synchronization

A large scale web crawler is a resource expensive application. Two large-scale web crawlers have been developed in a research setting: Google prototype [16] which was the first large-scale crawler developed at Stanford; and Mercator

[41, 51] developed at Compaq Systems Research Center. These crawlers were used to build and maintain the data repositories of the web for large scale search engines. In developing Mercator, Heydon et al. [41] found the DNS resolving and network bandwidth is the major bottleneck of crawler performance. Our crawler is built on mid-scale web domain where the number of hosts is significantly smaller. We use the internal DNS cache in LibWWW library to reduce the DNS resolving traffic. Crawling politeness policy and crawling traps are also discussed by Heydon et al. [41]. In general, Mercator sends hundreds requests per second but at most one request to a single server per second. Since our crawler runs on a limited number of hosts, it crawls web pages more aggressively to web servers, averaging 2.5 requests per second to a single server. Such crawling rate is appropriate for the development of domain level crawling. Najork and Wiener [52] examined the average page quality over time of pages downloaded during a web crawl using Mercator. They use the connectivity-based metric PageRank to measure the quality of pages and show that traversing the web graph in breadth-first search order is a practically good crawling strategy. Cho et.al [26] compared four different URL frontier ordering metrics: random ordering, breath-first ordering, backlink count ordering and PageRank value ordering. It demonstrates that if the basic unbiased crawling algorithm is used, the PageRank metric is the best one in ordering the URL frontier to crawl high quality web pages. Crawlers designed to crawl relevant pages to a given topic are often referred as Focused Crawler [21, 48, 20] or Topic-driven Crawler [49]. Focused crawlers use heuristics on choosing links to follow to try to gather pages with particular characteristics. Focused crawling algorithms often use similarity metric to analyze the text of URL or the anchor text of that link [26]. The URLs whose URL string or the anchor text contains some given

keywords are chosen to insert into the crawling frontier.

In order to maintain the freshness of the local web index, incremental web crawlers are used to repeatedly download web pages. A number of works [22, 28, 23, 24] studied the problems of how to select web pages to re-crawl and how to manage the crawling set. Given the change frequencies of web pages and a fix amount of crawling resources, Cho and Garcia-Molina [22, 24] modeled the problem as an optimization problem that the freshness of the web index is the goal while the resources are constrains. They showed that the optimal scheduler can be obtained through the method of Lagrange multipliers.

5.4 Hypertext information extraction

The process of web information extraction can be viewed as mining particular information from text and hypertext. It is well known that many business web sites use templates to generate web pages. Such templates often contain business logo, copyright, web master information, and many useful navigational links to top level resources within the same business domain. Templates help structure web content and make the web pages within the same site share the same look and feel. The informative content on the page is embedded within these templates other than provided by pure text. Often, the web pages in a business domain are generated by programs using the back end data repository. Computer programs can easily build template oriented web pages. For example, templates are frequently used by industry (e.g. Amazon.com) to present information of products, services, etc. Based on such observations, recently, there are several interesting works studying the methods of extracting useful information on web pages using the structural view of web documents

[66, 38, 60, 61, 56, 57, 64, 10, 65]. Most of these works are based on the static view of web pages, and often based on a tree-structured view of HTML documents. The essential goal of these approaches is to find informative content from Web documents. Zhai and Bin [66] used document tree alignment to find product information on web pages. A dynamic programming algorithm is proposed to match web data and extract product information that shares common tree structure. The complexity of the algorithm is $O(N^2)$. Dynamic programming matches nodes bottom-up while our tree matching algorithm is top-down. Bottom-up approach is appropriate when the bottom-level structure of the document tree remains relatively static, e.g. the product information on commercial web sites. However, top-down approach is appropriate when the bottom level structure changes a lot. As we showed, new content on web pages is often added at low levels. Therefore a top-down matching is chosen in our approach. Top-down matching has a superior performance too. Gupta et al. [38] gave an approach of content extraction based on the DOM-structure of HTML documents [1]. They provide a set of rules for content filtering to remove clutter (such as pop-up ads, unnecessary images and extraneous links) around the body of a web article that distracts a user from actual content. Yi et al. [64] proposed an cooperative HTML block filtering algorithm using data mining to eliminate noisy information on business web pages such as navigation panels, copyright and privacy notices, and advertisements, etc. It first builds a style tree of a set of similar web documents which is the combination of the DOM trees of these documents. Then the algorithm examines the complexity of the presentation styles in each sub tree of the style tree to eliminate the most common ones. Extracting information using the methods of tree edit distance [57] and machine learning [61, 60] are discussed recently. Reis et al. [57]

used algorithms of tree edit distance to extract common templates of news web site. Song et al. [61] used learning methods to automatically assign importance scores to pre-extracted blocks on web pages. Shih and Karger [60] used URL and table layout in the tree structure of web pages to extract news content from news web site. It provides a learning approach for automatic Web page classification tasks such as content recommendation and ad blocking. Bar-Yossef and Rajagopalan [10] propose the concept of *Pagelet* which is a self-contained logical region within a page that has a well defined topic or functionality. Pagelets can often be grouped in terms of templates which may be shared by multiple pages in a web site. By detecting and removing common templates, the filtered web pages are more likely to carry informative content and therefore can be used to improve the performance of web ranking and focused crawling. Other than extracting important content on Web documents, the HTML tree structures are also used for document segmentation [65, 56]. All these works are closely related to our design of extracting web changes from different versions of web documents.

5.5 Web ranking

Query independent evaluation of web documents often uses the link structure of the Web. Such link analysis can infer which pages are mostly referenced and visited in the web. The best known of these are the PageRank algorithm [55] and the HITS algorithm [43] as we discussed in Section 5.1. Both of them are discussed for years and a number of improved algorithms have been proposed [53, 46, 58, 47, 19, 40, 33, 32]. Ng et al. [53] discussed the stabilities of PageRank and HITS algorithms under perturbations to the link structure.

Lempel and Moran [46] identifies the Tightly Knit Community (TKC) Effect in HITS ranking that HITS algorithm often over-ranks tightly connected web page collections. They introduced SALSA, an improved HITS algorithm with link normalization. Li et al. [47] improves HITS by weighting links with relevance score. To combine content information and link analysis together to rank documents, Richardson and Domingos [58], Chakrabarti et al. [19] and Haveliwala [40] propose several effective solutions. Ding et al. [33] builds a unified framework that could represent PageRank, HITS in the same frame and indicates that the ranking results of them agree with each other quite well. Diligenti et al. [32] uses a different notation combining PageRank and HITS into a single ranking system. Different ranking results can be combined together to produce more effective ranking using ranking aggregation method [34]. In a addition to these, our link-based algorithm is designed to alleviate the local aggregation problem which has not been addressed before. Recently, Del Corso et al [29] studied the problem of how to rank news article. They proposed a ranking framework which models: (1) the process of generation of a stream of news articles, (2) the news articles clustering by topics, and (3) the evolution of news story over the time. Their solution for evaluating the evolution of news stories is similar to our approach of ranking the novelty of web changes.

Chapter 6

Conclusion

Web search engines usually maintain a huge index and update it slowly. In order to bring new information to users in a timely manner, we studied the problem of incremental web search by tracking changes of web documents. Incremental web search requires a much smaller amount of data processing than full indexing of the web. Therefore, new information carried by changes can be updated in the web index more quickly. There are several key problems in incremental web search: detecting changes, extracting changes, evaluating changes and modeling changes. In this thesis, we developed effective solutions to these problems. First, we found that detecting changes using HTTP meta data can successfully reduce network traffic by about 67%. Second, existing algorithms for extracting changes using tree edit distance have high computational cost, which is inappropriate for large-scale search engine development. We proposed a new algorithm, which reduces the cost to linear using both tree encoding and level-by-level tree matching. Our algorithm is appropriate for extracting changes between different versions of web pages. Third, the evaluation of changes differs from the evaluation of web documents. A unified ranking

framework is proposed combining three ranking metrics: popularity ranking, content-based ranking and evolution ranking. Fourth, we modeled web changes using survival analysis. We successfully discovered that the PageRank score is a good predictor of change frequencies of web pages. Such predictor can be combined with change history date of web pages to improve the effectiveness of the estimator of change frequencies.

Using the techniques we proposed, we developed an web search application, named “Web Daily News Assistant (WebDNA): finding what’s new on Your Web”, which helps community users search for new information on their community web. We present the framework, architecture and components of this application. We also discussed the techniques that are useful for crawling web pages on a medium scale web. Currently WebDNA is deployed on New York University web site. It provides several services for community web users: news digest university wide, news digest for different departments, change history for each single web page and full-text search on web changes.

There are several future directions for our studies:

- Our study focuses on the new information appearing in the changes on existing web pages. How to search for information posted on new pages is not well studied yet. When a new page is created, it is often referenced by a new link on a existing page. A short summary is often presented around the link. Tracking changes can successfully retrieve the summary but not the complete content of the newly created page. A good future direction is to integrate both changed content and newly created content into a unified search index. Such index can have a higher coverage of new information on the web.

- We learn that time related information appears at high frequency in new information and can be used to evaluate the quality. However a precise extraction of time information has not been studied yet. It is interesting to develop language patterns for the time information on web pages to evaluate the timeliness of new information more precisely.
- As we learned, popular pages are more likely to be updated by web authors. A further thought is that whether updates on popular pages are of high quality for retrieval. Cho et al. [27] proposed the concept of page quality, which is closely related to the popularity metrics of web pages. In our prospection, if such quality metric can be used to evaluate the updates of web pages, then people may be able to develop solutions to improve the quality, as well as freshness, of web index for retrieval. We leave it as a good future direction that how different metrics (change frequency, quality, popularity, etc.) can be used in a unified framework for web index synchronization problem.

Appendix A

Estimating the number of distinct trees

Consider we have M nodes spreading in K non-overlapping C -trees (each non-leaf node has C child nodes) of depth $d = 1, 2, 3, \dots, D$. Suppose the probability that a node locates in a tree of any height d is the same, denoted as Φ . Let n_d be the number of trees of depth d . Here we describe how we estimate the number K here.

We know a complete tree having depth d has $\frac{C^d - 1}{C - 1}$ nodes. Then we have the followings holds:

$$\sum_{d=1}^D n_d \cdot \frac{C^d - 1}{C - 1} = M \quad (\text{A.1})$$

$$\sum_{d=1}^D n_d = K \quad (\text{A.2})$$

$$\sum_{d=1}^D \frac{n_d \cdot \frac{C^d - 1}{C - 1}}{M} = \Phi \quad (\text{A.3})$$

From Eq. A.3 we get $n_d = \frac{\Phi M(C-1)}{C^d - 1}$, replace it in Eq. A.2 and get the following

$$\Phi = \frac{K}{M(C-1)} \cdot \frac{1}{\sum_{i=1}^D \frac{1}{C^{i-1}}}$$

Combined with Eq. A.3, we have

$$n_d = \frac{K}{C^d - 1} \cdot \frac{1}{\sum_{i=1}^D \frac{1}{C^{i-1}}}$$

Finally we get the mathematical representation of K

$$K = \frac{M(C-1)}{D} \cdot \sum_{i=1}^D \frac{1}{C^i - 1} \quad (\text{A.4})$$

First we know $\sum_{i=1}^D \frac{1}{C^{i-1}} > \frac{1}{C-1}$, then $K > \frac{M}{D}$.

Second, we know $\frac{1}{C^{i-1}} < \frac{1}{C^{i-1}}$ for $i > 1$. So we get

$$\begin{aligned} \sum_{i=1}^D \frac{1}{C^{i-1}} &< \frac{1}{C-1} + \sum_{i=1}^{D-1} \frac{1}{C^i} \\ &= \frac{1}{C-1} + \frac{1 - \frac{1}{C^D}}{C-1} \\ &< \frac{2}{C-1} \end{aligned}$$

Thus $K < \frac{2M}{D}$. Now we obtained the estimation of K as

$$\frac{M}{D} < K < \frac{2M}{D}$$

Appendix B

Computing minimum backward distance

As we are only interested in computing minimal backward distance of neighboring pages and the web graph is sparse, we present an algorithm which can compute MinBD much more efficiently than computing the all-pair shortest paths.

For simplicity, we assume the web graph is a directed unweighted graph without any link from and to the same page. Our algorithm starts with a depth-first-search (DFS), repeatedly updates the minimal backward distance in the DFS when new cycles are found. We color each graph node with WHITE, GREY and BLACK representing whether it is *unvisited*, *visited but not finished* and *finished* in DFS. For each page i , the algorithm stores a list of *cycle ancestors*, each of which locates in the DFS path from root page to page i and there exists a cycle containing the *cycle ancestor* and i . And for each *cycle ancestor* of a page, the algorithm stores the minimal distance from the *cycle ancestor* to that page in convenience of future update of MinBD in DFS. The algorithm picks a

root page and runs recursively on the following procedure:

```
1 MinBD_DFS = (page $i$)
2 {
3   Color page i as GREY
4   for each link i->j
5   {
6     if page j is WHITE    // tree link
7       MinBD_DFS(j)
8     else if page j is GREY // back link
9       UPDATEDFSPATH(j, i)
10    else if page j is BLACK // cross link
11    {
12      for each cycle ancestor k of page j do
13      {
14        if k is GREY
15          UPDATEDFSPATH(k, i)
16      }
17    }
18  }
19  Color page $i$ as BLACK
20 }
```

where the procedure *UPDATE_DFSPATH()* is defined as

```
1 UPDATEDFSPATH (page i, page j)
2 {
3   for each link k->l along DFS path i to j
4   {
```

```

5     update MinBD of link k->l
6     store cycle ancestor i in node l
7     update minimal distance from l to i
8   }
9 }

```

The correctness of this algorithm comes from the following lemmas:

Lemma 7. *In DFS, if a back link (a link pointing to a GREY node) is found, a new cycle is found.*

Lemma 8. *In DFS, if an cross link (a link pointing to a BLACK node) $i \rightarrow j$ is found, there exists a cycle containing this link if and only if there is a cycle containing j with ancestor of a GREY node.*

The first lemma is obvious. We state the proof of the second one here:

Proof. For an inter link $i \rightarrow j$, if there is a cycle containing j with ancestor of a GREY node k , then there exists a DFS path from k to i , denoted as $k \rightsquigarrow i$. Denote the arc from j to k in the cycle containing j and k as $j \rightsquigarrow k$. We see $i \rightarrow j$, $j \rightsquigarrow k$ and $k \rightsquigarrow i$ is a cycle containing the link $i \rightarrow j$. On the other hand, if there exists a circle C within all the links found so far in DFS, which contains link $i \rightarrow j$, C must contain at least one GREY node because the page i is the current node in DFS and it must be linked by its parent, which is a GREY node. Furthermore, the nodes with minimum DFS depth in such a cycle must also contain a GREY node. Otherwise, there exists a link on this cycle satisfying that it links from a BLACK node to a GREY node and the BLACK node has smaller DFS depth. This contradicts with the property of DFS. Proof done. □

Appendix C

Top frequent words in the web pages and web changes of the NYU web site

Time information	Time related	University Related	Popular Topics	Misc
2005	new	nyu	adult	information
pm	event	york	online	public
2004	news	research	free	writer
july	press	university	poker	reporter
june	now	student	sex	editor
april	recent	school	people	special
october		center	case	posted
march		program	casino	note
year		study	blackjack	comments
september		journalism	video	street
tuesday		service	work	download
november		faculty	american	content
2003		education	perl	
week		department	gay	
august		system	life	
today		policy	game	
		office	law	
		course	blog	
			health	
Popular topics continued				
social media world art risk market home state international web business				
teen internet city music national development movie data server story				
women search girls history community science security technology				

Figure C.1: Top 100 most frequent words in web changes of NYU Web.

Time information	Time related	University Related	Popular Topics	Misc
2004	newdate	nyu	home	information
2005	current	york	field	file
2003	events	university	search	share
2001	news	research	service	index
year		program	data	value
2002		school	contact	public
		student	work	type
		center	medicine	page
		department	world	number
		faculty	medical	description
		course	protein	general
		study	history	length
		education	node	
		graduate	copy	
		office	art	
		seminar	state	
Popular topics continued				
american law directory java note group case section science international web slide social business line institute management development life man mail support function code resource form example source project analysis room record policy software process method print system health package point people title version market				

Figure C.2: Top 100 most frequent words in the NYU Website.

Appendix D

Estimating the lifetime of web changes on a single web page

Given a web page W and its average lifetime of versions, T_{ver} . Here we show how to estimate the average lifetime of web changes, denoted as T_{chg} , on W . Suppose the probability that a web change is removed by a new version of W is $p = 0.5$. At time t_0 , a web change C is created. The the probability that it is removed at time kT_{ver} is $(1 - p)^{k-1}p$. The expected lifetime of C is given as follows:

$$T_{chg}(C) = \sum_{k=1}^{\infty} kT_{ver} \cdot (1 - p)^{k-1}p \quad (D.1)$$

Eq. D.1 - Eq. D.1 multiplied by $1 - p$, we get

$$\begin{aligned} & T_{chg}(C) - (1 - p)T_{chg}(C) \\ &= T_{ver}(1 - p) + \sum_{k=2}^{\infty} T_{ver}(1 - p)^{k-1}p \\ &= T_{ver}p + T_{ver}(1 - p) \\ &= T_{ver} \end{aligned}$$

Therefore

$$T_{chg}(C) = \frac{T_{ver}}{p} = 2T_{ver} \quad (\text{D.2})$$

Appendix E

The probabilities for sampling web pages

Let $f(\ln PR)$ be the density function of the logarithm of PageRank values, the size of sample set be K , and the probability of picking a page with PageRank score PR into the sample set be $Prob[PR]$. An evenly-distributed sampling for all PageRank scores satisfies the following:

$$Prob[PR]df(\ln PR) = \frac{K d \ln PR}{\int_{\min(\ln PR)}^{\max(\ln PR)} d \ln PR} = \frac{K}{\max(\ln PR) - \min(\ln PR)} \quad (\text{E.1})$$

From Figure 4.2 (a) we know that

$$\frac{d \ln f(\ln PR)}{d \ln PR} = \frac{df(\ln PR)}{f(\ln PR)d \ln PR} = \text{constant} \quad (\text{E.2})$$

Combine Eq. E.1 and Eq. E.2, we get

$$Prob[PR] = \frac{K}{C_0(\max(\ln PR) - \min(\ln PR))f(\ln PR)} \propto \frac{1}{f(\ln PR)} \quad (\text{E.3})$$

Bibliography

- [1] Document Object Model (DOM). <http://www.w3.org/DOM/>.
- [2] Google News. <http://news.google.com>.
- [3] HTML 4.01 Specification. <http://www.w3.org/TR/html4/>.
- [4] LibWWW - the W3C Protocol Library. <http://www.w3.org/Library/>.
- [5] The R project for statistical computing. <http://www.r-project.org/>.
- [6] RFC 1321 - The MD5 Message-Digest Algorithm. <http://www.faqs.org/rfcs/rfc1321.html>.
- [7] RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. <http://www.faqs.org/rfcs/rfc2616.html>.
- [8] R. Albert, A.-L. Barabasi, and H. Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A*, 281, 2000.
- [9] A. Arasu, J. Cho, and H. Garcia-Molina. Searching the web. *ACM Transactions on Programming Languages and Systems*, 1(1), 2001.

- [10] Z. Bar-Yossef and S. Rajagopalan. Template detection via data mining and its applications. In *Proc. 11th International World Wide Web Conference*, 2002.
- [11] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 1999.
- [12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. 18th IEEE INFOCOM The Conference on Computer Communications*, 1999.
- [13] B. Brewington and G. Cybenko. How dynamic is the web. In *Proc. 9th International World Wide Web Conference*, 2000.
- [14] B. Brewington and G. Cybenko. Keeping up with the changine web. *IEEE Computer*, 33(5), 2000.
- [15] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proc. 1995 ACM SIGMOD Special Interest Group on Management of Data*, 1995.
- [16] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. 6th International World Wide Web Conference*, 1998.
- [17] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proc. 9th International World Wide Web Conference*, 2000.
- [18] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proc. 6th International World Wide Web Conference*, 1997.

- [19] S. Chakrabarti, B. Dom, D. Gibson, J. Kleinberg, P. Raghavan, and S. Rajagopalan. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proc. 7th International World Wide Web Conference*, 1998.
- [20] S. Chakrabarti, K. Punera, and M. Subramanyam. Accelerated focused crawling through online relevance feedback. In *Proc. 11th International World Wide Web Conference*, 2001.
- [21] S. Chakrabarti, M. van de Berg, and B. Dom. Focused crawling: a new approach to topic-specific web resource discovery. In *Proc. 8th International World Wide Web Conference*, 1999.
- [22] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. 26th International Conference on Very Large Data Bases*, 2000.
- [23] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proc. 2000 ACM SIGMOD Special Interest Group on Management of Data*, 2000.
- [24] J. Cho and H. Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems*, 28(4), 2003.
- [25] J. Cho and H. Garcia-Molina. Estimating frequency of change. *ACM Transactions on Internet Technology (TOIT)*, 2003.
- [26] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. In *Proc. 7th International World Wide Web Conference*, 1998.

- [27] J. Cho and S. Roy. Impact of search engines on page popularity. In *Proc. 13th International World Wide Web Conference*, 2004.
- [28] E. Coffman, J. Z. Liu, and R. R. Weber. Optimal robot scheduling for web search engines. *Technical report, INRIA*, 1997.
- [29] G. M. D. Corso, A. Gullf, and F. Romani. Ranking a stream of news. In *Proc. 14th International World Wide Web Conference*, 2005.
- [30] D. Curbera and A. Epstein. Fast difference and update of xml documents. In *XTech99*, 1999.
- [31] D. Dhyani, W.-K. Ng, and S. S. Bhowmick. A survey of web metrics. *ACM Computing Surveys*, 34(4), 2002.
- [32] M. Diligenti, M. Gori, and M. Maggini. Web page scoring systems for horizontal and vertical search. In *Proc. 11th International World Wide Web Conference*, 2002.
- [33] C. Ding, X. He, P. Husbands, H. Zha, and H. Simon. PageRank, HITS and a unified framework for link analysis. In *Proc. 25th ACM SIGIR Special Interest Group on Information Retrieval*, 2002.
- [34] C. Dwork, S. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proc. 10th International World Wide Web Conference*, 2001.
- [35] D. Fallows, L. Rainie, and G. Mudd. The popularity and importance of search engines, The Pew Internet and American Life Project. http://www.pewinternet.org/pdfs/PIP_Data_Memo_Searchengines.pdf, 2004.

- [36] D. Fretterly, M. Manasse, M. Najork, and J. Wiener. A large-scale study of the evolution of web pages. In *Proc. 12th International World Wide Web Conference*, 2003.
- [37] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 2003.
- [38] S. Gupta, G. Kaiser, D. Neistadt, and P. Grimm. DOM-based content extraction of HTML documents. In *Proc. 12th International World Wide Web Conference*, 2003.
- [39] T. Haveliwala. Efficient computation of PageRank. *Stanford Digital Library working paper*, 1999.
- [40] T. Haveliwala. Topic-sensitive PageRank. In *Proc. 11th International World Wide Web Conference*, 2002.
- [41] A. Heydon and M. Najork. Mercator: a scalable, extensible web crawler. In *Proc. 8th International World Wide Web Conference*, 1999.
- [42] P. G. Ipeirotis, A. Ntoulas, J. Cho, and L. Gravano. Modeling and managing content changes in text databases. In *Proc. 21st IEEE International Conference on Data Engineering (ICDE)*, 2005.
- [43] L. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998.
- [44] R. Kosala and H. Blockeel. Web mining research: a survey. In *Proc. 6th ACM SIGKDD Special Interest Group on Knowledge Discovery and Data Mining*, 2000.

- [45] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for cyber communities. In *Proc. 8th International World Wide Web Conference*, 1999.
- [46] R. Lempel and S. Moran. The stochastic approach for link-structure analysis (SALSA) and the TKC effect. In *Proc. 9th International World Wide Web Conference*, 2000.
- [47] L. Li, Y. Shang, and W. Zhang. Improvement of HITS-based algorithms on web documents. In *Proc. 11th International World Wide Web Conference*, 2002.
- [48] M. Diligenti, F. Coetzee, S. Lawrence, C. Giles, and M. Gori. Focused crawling using context graph. In *Proc. 26th International Conference on Very Large Data Bases*, 2000.
- [49] F. Menczer, G. Pant, P. Srinivasan, and M. Ruiz. Evaluating topic-driven web crawlers. In *Proc. 24th ACM SIGIR Special Interest Group on Information Retrieval*, 2001.
- [50] E. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2), 1986.
- [51] M. Najork and J. Wiener. High-performance web crawling. *SRC Research Report 173, Compaq System Research Center*, 2001.
- [52] M. Najork and J. L. Wiener. Breadth-first search crawling yields high-quality pages. In *Proc. 10th International World Wide Web Conference*, 2001.

- [53] A. Ng, A. Zheng, and M. Jordan. Stable algorithms for link analysis. In *Proc. 24rd ACM SIGIR Special Interest Group on Information Retrieval*, 2001.
- [54] A. Ntoulas, J. Cho, and C. Olston. What's new on the web?: the evolution of the web from a search engine perspective. In *Proc. 13th International World Wide Web Conference*, 2004.
- [55] L. Page, S. Brin, R. Motowani, and T. Winograd. The PageRank citation ranking: bringing order to the web. *Stanford Digital Library working paper*, 1997-0072, 1997.
- [56] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglass. Automatic detection of fragments in dynamically generated web pages. In *Proc. 13th International World Wide Web Conference*, 2004.
- [57] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender. Automatic web news extraction using tree edit distance. In *Proc. 13th International World Wide Web Conference*, 2004.
- [58] M. Richardson and P. Domingos. The intelligent surfer: probabilistic combination of link and content information in PageRank. In *Proc. Advances in Neural Information Processing Systems*, 2002.
- [59] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, first edition edition, 1983.
- [60] L. K. Shih and D. R. Karger. Using URLs and table layout for web classification tasks. In *Proc. 13th International World Wide Web Conference*, 2004.

- [61] R. Song, H. Liu, J.-R. Wen, and W.-Y. Ma. Learning block importance models for web pages. In *Proc. 13th International World Wide Web Conference*, 2004.
- [62] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *Proc. 19th IEEE International Conference on Data Engineering*, 2003.
- [63] Z. Wang. Improved link-based algorithms for ranking web pages. In *Proc. 5th International Conference of Web Age Information Management*, 2004.
- [64] L. Yi, B. Liu, and X. Li. Eliminating noisy information in web pages for data mining. In *Proc. 9th ACM SIGKDD Special Interest Group on Knowledge Discovery and Data Mining*, 2003.
- [65] S. Yu, D. Cai, J.-R. Wen, and W.-Y. Ma. Improving pseudo-relevance feedback in web information retrieval using web page segmentation. In *Proc. 12th International World Wide Web Conference*, 2003.
- [66] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proc. 14th International World Wide Web Conference*, 2005.
- [67] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18-6, 1989.
- [68] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42, 1992.