CONTEXT-SENSITIVE WEB SEARCH

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Taher H. Haveliwala
May 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christopher D. Manning

Approved for the University Committee on Graduate Studies.

Abstract

As the Web continues to grow and encompass broader and more diverse sources of information, providing effective search facilities to users becomes an increasingly challenging problem. To help users deal with the deluge of Web-accessible information, we propose a search system which makes use of *context* to improve search results in a scalable way. By context, we mean any sources of information, in addition to any search query, that provide clues about the user's true information need. For instance, a user's bookmarks and search history can be considered a part of the search context.

We consider two types of context-based search. The first type of functionality we consider is "similarity search." In this case, as the user is browsing Web pages, URLs for pages similar to the current page are retrieved and displayed in a side panel. No query is explicitly issued; context alone (i.e., the page currently being viewed) is used to provide the user with useful related information. The second type of functionality involves taking search context into account when ranking results to standard search queries.

Web search differs from traditional information retrieval tasks in several major ways, making effective context-sensitive Web search challenging. First, scalability is of critical importance. With billions of publicly accessible documents, the Web is much larger than traditional datasets. Similarly, with millions of search queries issued each day, the query load is much higher than for traditional information retrieval systems. Second, there are no guarantees on the quality of Web pages, with Web-authors taking an adversarial, rather than cooperative, approach in attempts to inflate the rankings of their pages. Third, there is a significant amount of metadata embodied in the link structure corresponding to the hyperlinks between Web pages that can be exploited

during the retrieval process. In this thesis, we design a search system, using the Stanford WebBase platform, that exploits the link structure of the Web to provide scalable, context-sensitive search.

Acknowledgments

I would like to thank my advisor, Jeffrey Ullman, for his years of guidance during my graduate studies. His insightful comments and suggestions greatly influenced my research, and helped turn ideas into reality.

I would like to thank Hector Garcia-Molina and Christopher Manning for serving on my reading and oral defense committees. I am fortunate to have had the privilege of working with them and their students throughout my years at Stanford.

Special thanks to Andreas Paepcke, Andy Kacsmar, and Gary Wesley for their tireless commitment to the Stanford WebBase project. I am fortunate to have been able to collaborate so closely with Wang Lam, Sriram Raghavan, and Junghoo Cho on WebBase development.

I would also like to thank my coauthors and colleagues from whom I have learned so much. In addition to the aforementioned, I have had the great privilege of collaborating with, and learning from, Gene Golub, Sepandar Kamvar, Glen Jeh, Dan Klein, Aristides Gionis, and Piotr Indyk. Discussions with Rajeev Motwani early in my graduate studies were also influential to my work. In addition, I have learned a great deal over the years from the many members of the Stanford database group; the atmosphere of cross communication and collaboration within the database group made my years at Stanford both enjoyable and immensely rewarding.

I also thank my family for their unwavering support in all of my endeavors. My parents' selfless choices throughout my life have helped me reach this point.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Web search engines have become indispensable tools for users seeking information on the Web. From the perspective of a typical user, a Web search engine is a service that allows them to enter a short piece of text (the *search query*) that describes an information need; the search engine then returns to the user a list of Web pages (the *search results*) ordered by how likely they are to satisfy the user's information need. For instance, a user who wants to learn more about light bulbs might go to a search engine and type in "light bulb," and click on the top few results to find relevant information.

A Web search engine consists of several components that are necessary to support this seemingly simple interaction:

> **Crawler:** The crawler is responsible for finding, collecting, and storing as many publicly accessible Web pages as possible. The collection of pages stored on the search engine's server is called a *Web-page repository*. Crawlers face numerous challenges, including how to select which pages to crawl, how to efficiently fetch billions of pages, and how to minimize the impact on the servers hosting these pages [17].

Text Indexer: The text indexer is responsible for processing the Web-page repository collected by the crawler to build an *inverted text index* that allows quickly determining which documents in the repository contain a particular word. Although text indexers are conceptually simple, scaling them to billions of pages is a challenging task [55, 74].

Auxiliary Indexers: Auxiliary indexers are responsible for constructing additional indexes that provide metadata about the pages in the repository, used for ordering search results by how likely they are to satisfy a user's information need. For instance, an auxiliary indexer might tell us the length of a page or how popular the page is based on a chosen measure of popularity. These auxiliary indexers can vary dramatically in their complexity.

Query-Processor: The query-processor is the component responsible for accessing the text and auxiliary indexes in response to a search query, and constructing a ranked listing of Web pages likely to satisfy the information need underlying the query. Designing rankings algorithms to order pages by their relevance to the user's information need is one of the crucial challenges Web search engines face.

The earliest Web search engines relied primarily on matching up the words in Web pages and the words in the search query to retrieve and rank the search results. However, this approach was unable to deal effectively with the dramatic growth of the Web; search results to most queries yield thousands or even millions of Web pages, making it difficult to determine which ones to display first using only text-based cues. In general, a user may only have the patience to view the top 10 results returned. Furthermore, a malicious practice known as *keyword spamming* quickly arose — Webmasters who wanted their pages to appear at the top of the search results for a particular query would simply repeat the words in that query many times in their pages, making them appear relevant to purely text-based ranking algorithms.

To improve upon these algorithms, search engines began exploiting the *hyperlink structure* of the Web to rank search results [8, 27]. A Web page has the ability to refer to another page with a *hyperlink*, which consists of the URL of the target page as well

as a piece of text called *anchor-text* that describes the target page. These hyperlinks, or simply *links*, can be viewed as inducing a graph structure over the Web, where the nodes of the graph correspond to Web pages, and the edges of the graph correspond to links between pages. Researchers began exploiting this graph structure for improving search result rankings; one of the earliest algorithms for doing so was PageRank [61], which analyzed the Web graph to assign to each Web page a query-independent notion of importance or popularity. By analyzing the hyperlink structure of the Web, and constructing auxiliary indexes that capture the link-popularity of pages, search engines were better able to rank search results in a spam-resistant way.

Despite numerous improvements in algorithms for ranking Web search results, search engines have generally relied solely on the search query to determine a user's information need. As the Web continues to grow, and as the diversity of users increases, we believe search engines must utilize contextual clues to help satisfy users' information needs. Context-sensitive search, which takes into account the interests of the user as well as the specific context in which the search was issued, is the next step in providing users with the most relevant information possible. Context refers to any sources of information, in addition to the query itself, that provide clues about the user's true information need. For instance, the history of queries issued leading up to the current query is a form of query context. A search for "basketball" followed up with a search for "Jordan" presents an opportunity for disambiguating the latter query. The first query "basketball" is evidence that at the time the second query "Jordan" was issued, the user is looking for information about the athlete Michael Jordan, not about the country Jordan. As another example, a user's bookmarks provide clues about their interests, and thus contributes to the search context. We propose a scalable search system which makes use of *search context* to improve the search experience.

1.2 Dissertation Overview

In this dissertation, we design and implement a system capable of supporting context-based search using the Stanford WebBase platform. We consider two scenarios. In the

first scenario, we assume a user is browsing the Web to satisfy an information need; our system can display URLs for pages similar to the page being viewed. This scenario represents a purely context-based search — no query is explicitly issued by the user. Context alone (e.g., the page the user is currently viewing) is used by the system to provide useful related information that may help satisfy the user's information need. In the second scenario, we assume a user with an information need explicitly issues a keyword search query to our system. Our system exploits search context to influence the search result rankings for that query. This dissertation is organized as follows:

Chapter 2: Similarity Search

The functionality to support the first context-based search scenario described above is known as *similarity search*. Given a query URL, a similarity-search system should be able to quickly retrieve a ranked list of URLs for other pages that are in some way similar to the query URL. The key challenges in designing a similarity-search system are developing the right notion of Web page similarity, and scaling the system to support efficient query processing over large repositories.

The primary contributions of this chapter can be summarized as follows. We introduce a technique for evaluating different similarity-search strategies by comparing them with the similarity judgments embodied in human-constructed Web directories such as Yahoo! or the Open Directory [60, 75]. We explore a large number of strategies to determine which one performs the best under our evaluation technique. In particular, we found that among the sources of information considered, the information contained in the anchor-text describing target pages is the strongest source of similarity information. Finally, we describe the design and implementation of a large-scale similarity-search system that employs the best-performing similarity-search strategy.

Chapter 3: Topic-Sensitive Search

Beginning with Chapter 3, we consider the second context-based search scenario described earlier. In this chapter, we introduce the core of our context-sensitive keyword search system, which makes use of a set of topics to represent the context associated

with the search query. We describe how we extend the PageRank link-analysis ranking algorithm to account for search context in a practical and scalable way by making the computation topic sensitive. The key idea in our system is that instead of assigning a single estimate of importance to each page, we assign several such estimates. We call our link-analysis algorithm *Topic-Sensitive PageRank*.

The primary contributions of this chapter include the Topic-Sensitive PageRank algorithm, the empirical results demonstrating how our algorithm affects search-result rankings, and an analysis of the theoretical properties of our algorithm in terms of matrix computations.

Chapter 4: Computing PageRank by Power Extrapolation and
Chapter 5: Block-Oriented PageRank Computation
Because our system requires running a variant of the PageRank algorithm multiple times when building the auxiliary page-ranking index for the repository, speeding up PageRank computations is an important challenge. In Chapters 4 and 5, we describe techniques for speeding up the PageRank computation on large datasets. In Chapter 4, we describe an algorithm for speeding up the computation of PageRank using a technique known as extrapolation. PageRank is an iterative computation; Power Extrapolation is a technique we developed to reduce the number of iterations required. In particular, we show that Power Extrapolation is able to speed up the computation of PageRank by 30%.

In Chapter 5, we take a different approach to speeding up the PageRank computation. In the first part of the chapter, we describe how to compute PageRank for large datasets in a memory efficient way, without making use of any special properties of the underlying hyperlink structure of the Web. In the second part of the chapter, we introduce the BlockRank algorithm that exploits the *nested block structure* of the Web graph to speed up the computation of PageRank. Nested block structure refers to a special linking pattern that is prevalent on the Web — pages tend to link to other pages with similar URL prefixes. E.g., the majority of outlinks from pages whose URLs are prefixed with www.stanford.edu/ are to other pages whose URLs are

prefixed with www.stanford.edu/.[1] We show that our BlockRank algorithm speeds up the computation of PageRank by a factor of 3.

Chapter 6: Efficient Encodings for Ranking Vectors
The topic-sensitive PageRank indexes that we compute can become very large, making it necessary to find ways of compressing them. In Chapter 6, we describe how we compress our PageRank indexes in a way that minimizes the impact on search result rankings. In particular, we describe how we use scalar quantization to lossily compress the ranking data, and describe how we measure any adverse effects of the compression by using a rank-based distortion metric. We show that we can compress the topic-sensitive ranking data by more than half without much loss in ranking precision.

1.3 Related Work

Here we give a brief overview of related work; more detailed discussions are given in the chapters that follow:

Similarity search: The "Related Pages" functionality provided by several major search engines represents related work. However, the details of these algorithms are not publicly available. Dean and Henzinger. [21] propose a related pages algorithm based purely on the connectivity of the Web graph (i.e., using only the edges of the Web graph, not anchor-text).

The idea of using anchor-text for document representation has been exploited in the past to attack a variety of other information retrieval problems [2, 8, 13, 14, 20, 48].

Link-analysis algorithms for search rankings: The first two link-analysis algorithms developed for information retrieval on the Web were *PageRank*, due

[1] As we will see in Chapter 5, the components of the hostname portion of the URL are generally reversed before comparing prefixes.

to Page et al. [61], and *Hyperlink-Induced Topic Search (HITS)*, due to Kleinberg [48, 49]. The PageRank algorithm computes a single value per page capturing the importance of each page using an iterative computation that propagates an initial estimate of page importance through the Web graph. The HITS algorithm computes two values for each page, the authority score and the hub score — the authority score captures the authoritativeness of a page, and the hub score captures the degree to which a page links to authoritative pages.

In recent years, there has been much research on extending these link analysis algorithms to make them more effective and on making the computations more efficient. Our work in personalized and context-sensitive extensions to PageRank and in techniques for computing PageRank more efficiently was among the earliest along this avenue of research. A survey of many of the approaches that have been proposed can be found in [7, 51].

Compressing ranking indexes: There has been much work on compressing the various indexes required for large-scale Web search, although the bulk of the attention has been directed towards compressing the text index (e.g., see Witten et al. [74]). There has been comparatively little work on the development of lossy encodings for compressing numeric ranking indexes (such as PageRank), the focus of our work in Chapter 6. The field of quantization [28] provides the framework for the encoding techniques we develop.

Part I

Similarity Search

Chapter 2

Similarity Search

2.1 Introduction[1]

In this chapter, we introduce the component of our context-sensitive search system that supports the ability to display to the user pages that may be of interest, using nothing more than a currently viewed page. This functionality boils down to similarity search — given a query page, a similarity-search system should return a list of "similar" Web pages. We say that two Web pages are similar if they are likely to satisfy the same kinds of information needs that users may have.

There are many possible approaches to determining whether Web pages are similar to one another; for instance, a high degree of *cocitation* might be an indicator of Web page similarity [21]. Two pages are cocited if a third page includes links to both of them. If two pages u and v are cocited frequently (i.e., there are many pages which include links to both of them), it signifies that many Web page authors describing some piece of information felt the need to refer to both u and v, making it reasonable to believe that u and v are similar. There are numerous other ways one might use to determine how similar Web pages are to one another — for instance, we might look at the overlap in the words used on the pages, or instead we might look at the overlap of the words in just the titles of the pages. Furthermore, there are many possible ways of measuring "overlap" — we might ignore how often words appear in the text,

[1]This chapter covers joint work we first presented in [38, 39]

or we might choose to make use of the frequency with which words appear. Each of these choices that we make for which indicators to use to estimate the similarity of Web pages leads to a different *similarity-search strategy*. Since there is no way for us to directly measure the property that two Web pages satisfy the same information needs, an important goal of our work is to select a similarity-search strategy that best captures that property.

Given a small number of possible similarity-search strategies, one might imagine comparing their relative quality through user studies, by asking humans how well a similarity-search strategy determines the "true" similarity of Web pages (e.g., how well a similarity-search strategy measures the degree to which Web pages satisfy the same information needs). However, user studies can have significant cost in both time and resources, especially if the number of possible similarity-search strategies is very large. In this situation, it is extremely desirable to automate strategy comparisons.

In this chapter, we develop an automated evaluation technique to choose a similarity-search strategy. In particular, we view manually constructed directories such as Yahoo! [75] and the Open Directory Project (ODP) [60] as implicit sources of human judgements about Web page similarity. For instance, if two pages are contained in the category `Arts/Music` in the ODP, then we know that a human felt those two pages were in some way similar. If these directories were complete, and categorized all Web pages, we might choose a directory-based similarity-search strategy that simply said that pages that are listed in the same category are similar to one another. However, these directories contain only a fraction of the pages on the Web — the ODP has fewer than 5 million pages, whereas the Web has over 8 billion pages. Therefore, we must come up with other kinds of strategies that work for all Web pages, such as measuring cocitation or textual overlap. Our approach to choosing among these possible similarity-search strategies is to measure how well each candidate strategy agrees with the directory-based strategy for those pages that *are* listed in the directory. We then use the strategy that agrees most closely with the directory-based strategy (on that small subset of pages) for measuring the similarity of *arbitrary* Web pages. We discuss the details of this evaluation technique in Section 2.2.

When considering candidate similarity-search strategies, we restrict our attention

to only those that measure the similarity of two Web pages by first representing each page as a multiset of terms and then measuring the overlap between the two multisets, using a particular measure for overlap we discuss later. Thus, the distinguishing characteristic of the candidate similarity-search strategies we consider is in exactly how they represent Web pages as multisets (or bags) of terms. There are many possible ways to represent a page using a bag of terms; for instance, we might choose to include the words that occur in the body of the page, or we may choose to include the words that occur in the text in the inbound links (i.e., anchor-text) from other pages used to refer to the page. We describe the choices involved in representing a page as a bag of terms in detail in Section 2.3, and describe the overlap measure we use for bags of terms in Section 2.4. In Section 2.5, we give our evaluation results for the various candidate strategies, and show which one performs the best.

Because of the potentially large number of terms that can get included in the multiset representing a page, it is nontrivial to efficiently construct a similarity index that allows quickly answering similarity-search queries. We discuss in Section 2.6 how a previously established technique based on a special kind of hashing can be used to implement similarity-search strategies that require multisets with large numbers of distinct terms. In particular, we develop an indexing approach relying on the Min-hashing technique [10, 18] to construct a similarity-search index for roughly 75 million pages to demonstrate the scalability of our approach. Because each stage of our algorithm is trivially parallelizable, our indexing approach can scale to the few billion accessible documents currently on the Web with a sufficient number of machines.

2.2 Evaluation Methodology

We describe next how we evaluate different similarity-search strategies using the similarity orderings implicit in human-built hierarchical directories. We use a hierarchical directory to induce sets of "ground truth" similarity orderings. Then, we compare the orderings produced by a particular similarity-search strategy to these ground truth orderings, using a statistical measure outlined below. We believe that strategies that yield higher values of this statistical measure will produce better results from the standpoint of a user of the system who is trying to satisfy an information need.

2.2.1 Finding a Ground-Truth Ordering

Unfortunately, there is no directly available ground truth in the form of either exact document-document similarity values or correct similarity search results, leading to the following problem:

Problem 1 SIMILARDOCUMENT **(notion of similarity)***: Formalize the notion of similarity between Web documents using an external quality measure.*

An *external quality measure* is a quality measure that makes of an external source of information — in our case, Web directories. There is a great deal of ordering information implicit in hierarchical Web directories. For example, a page u listed in the ODP `recreation/aviation/un-powered` class is more similar to other pages listed in that same class than to those outside of that class. Furthermore, page u is probably more similar to other documents in other classes under `recreation/aviation` than to pages entirely outside of that region of the directory. Intuitively, the pages most similar to u are the other documents in u's class, followed by those in sibling classes, and so on.

There are certainly cases where location in the hierarchy does not accurately reflect document similarity. Consider documents in `recreation/autos`, which are likely more similar to those in `shopping/autos` than to those in `recreation/smoking`. We have found that these cases are few enough that they do not have much effect on our evaluation criteria, since we average over the statistics of many documents.

To formalize the notion of distance from a source document to another document in the hierarchy we define *familial distance*.

Definition 1 *Let the* familial distance $d_f(s, d)$ *from a source document s to another document d in a class hierarchy be the distance from s's class to the most specific class dominating both s and d.*[2]

For simplicity, we collapsed the directory below a depth of three and ignored the (relatively few) documents above that depth. For example, we reassigned any pages listed under the category `Computers/Software/Graphics/Animation` to the

[2]We treated the hierarchy as a tree, ignoring the "soft-links" denoted by an "@" suffix

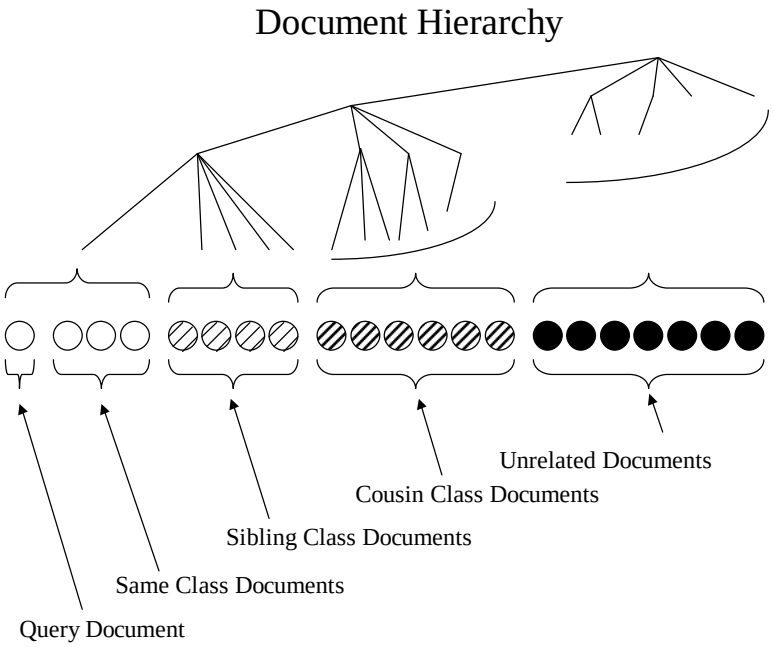


Figure 2.1: Mapping a hierarchy onto a partial ordering, given a source document.

category `Computers/Software/Graphics`. Therefore, there are only four possible values for familial distance, as depicted in Figure 2.1. We name these distances as follows:

Distance 0: *Same* – Pages are in the same class.

Distance 1: *Siblings* – Pages are in sibling classes.

Distance 2: *Cousins* – Pages are in classes which are first cousins.

Distance 3: *Unrelated* – The lowest common ancestor of the documents classes is the hierarchy root.

Given a source document, we wish to use familial distances to other documents to construct a partial similarity ordering over those documents. Our basic assumption is:

> *In general, the true similarity of documents to a source document decreases monotonically with the familial distance from that document.*

Given this principle, and our definition of familial distance, for any source document in a hierarchical directory, we can derive a partial ordering of all other documents in the directory. Note that we do not give any numerical interpretation to these familial distance values. We only depend on the above stated principle: a source document is on average more similar to a same-class document than to a sibling-class document, and is on average more similar to a sibling-class document than a cousin-class document, and so on.

Definition 2 *Let the* familial ordering $\prec_{d_f(s)}$ *of all documents with respect to a source document* s *be:* $\prec_{d_f(s)} = \{(a, b) |\ d_f(s, a) < d_f(s, b)\}$

This ordering is very weak, since for a given source, many pairs of documents are not comparable. For instance, given a page u in `Computers/Software/Graphics`, we cannot say which is more similar — a page v in `Computers/Software/Business` or a page w in `Computers/Software/Networking`. The majority of the distinctions that *are* made, however, are among documents that are very similar to the source and documents that are much less similar to the source. In contrast, no distinctions are made between pages in unrelated categories — using the previous example, we can say nothing of the similarity of page u to any page under the `Arts` branch vs. any page under the `Science` branch. However, we are generally not concerned with the degree to which we can make similarity judgements among pages that have very low similarity to the query page, so the familial ordering gives us ground truth judgements precisely for the cases we want.

2.2.2 Comparing Orderings

At this point, we have derived a partial ordering from a given hierarchical directory and query (source) document s that belongs in the hierarchy. We then wish to use this partial ordering to evaluate the correctness of an (almost) total ordering produced by our system.[3] Perhaps the most common method of comparing two rankings is the

[3] A strategy produces ties when two documents d_1 and d_2 have exactly the same similarity to the source document s. When this happens, it is nearly always because s has similarity 0 to both d_1 and d_2.

Spearman rank correlation coefficient, but this measure is best suited to comparing rankings with few or no ties [70]. A more natural measure for our requirements is the Kruskal-Goodman Γ [26]:

Definition 3 *For orderings* $\prec_a$ *and* $\prec_b$, $\Gamma(\prec_a, \prec_b)$ *is* $2 \times Pr[\prec_a, \prec_b$ *agree on* (x, y) , *given that* $\prec_a, \prec_b$ *both order* $(x, y)] - 1$

For instance, consider the orderings $\prec_1$: $(a, \{b, c\}, d)$ and $\prec_2$: (b, a, c, d). $\prec_1$ places a before the rest, and it places b and c before d. It makes no distinction between b and c. $\prec_2$ is a total ordering — it contains no ties. The two orderings agree on the relative order of a and d — they both say that a appears before d. The relative order of b and c are irrelevant when computing Γ, since $\prec_1$ does not order those two. The two orderings disagree on the relative order of a and b. Intuitively, there are a certain number of document pairs, and a given ordering only makes judgments about some of those pairs. When comparing two orderings, we look only at the pairs of documents that both orderings make a judgment about. A value of 1 is perfect accord, 0 is the expected value of a random ordering, and -1 indicates perfect reversed accord. We claim that if two rankings $\prec_a$ and $\prec_b$ differ in their Γ values with respect to a ground-truth $\prec_t$, then the ordering with the higher Γ will be the better ranking. However, there is a caveat here. We can make this claim only if $\prec_a$ and $\prec_b$ are both total orderings. If, say, $\prec_a$ is total, but $\prec_b$ only orders selected pairs, then the Γ value for $\prec_b$ will likely be artificially inflated. To correct for this inflation and make all of our settings comparable, we randomly break any ties in the rankings produced by our candidate similarity-search strategies to force all strategies to return a total order. Note that ties in the ground-truth familial ordering are not problematic, as they are constant and do not lead to inflated Γ values for any candidate strategy. Thus, we allow ties in the ground truth $\prec_t$, but not in orderings $\prec_a$ that are compared to the ground truth.

2.2.3 Regions of the Orderings

Thus, given a directory, a query document s, and a similarity strategy sim, we can construct two orderings over documents in the directory: the ground-truth familial

ordering $\prec_{d_f(s)}$, and the total ordering induced by our similarity strategy, $\prec_{sim(s)}$. We can then calculate the corresponding Γ value. This value gives us a measure of the quality of the ranking for that query document with respect to that similarity strategy and directory. However, we need to give a sense of how good our rankings are across all query documents, so we compute the average Γ value over a large number of test query pages.

In order to more precisely evaluate our results, however, we calculated three partial-Γ values that emphasized different regions of the familial ordering. Each partial-Γ is based on the fraction of correct comparable pairs *of a certain type*. Our types are:

***Sibling*-Γ:** Calculated using only pairs of documents (d_1, d_2) where d_1 is from the same class as the source document and d_2 is from a sibling class; i.e., we ignore all documents that are not either distance 0 or distance 1 from the source document when computing Γ.

***Cousin*-Γ:** Calculated using only pairs of documents (d_1, d_2) where d_1 is from the same class as the source document and d_2 is from a cousin class; i.e., we ignore all documents that are not either distance 0 or distance 2 from the source when computing Γ.

***Unrelated*-Γ:** Calculated using only pairs of documents (d_1, d_2) where d_1 is from the same class as the source document and d_2 is from an unrelated class; i.e., we ignore all documents that are not either distance 0 or distance 3 from the source when computing Γ.

These partial-Γ values allow us to inspect how various similarity measures performed on various regions of the rankings. For example, sibling-Γ performance indicates how well fine distinctions are being made near the top of the familial ranking, while unrelated-Γ performance measures how well coarser distinctions are being made. Unrelated-Γ being unusually low in relation to sibling-Γ is also a good indicator of situations when the top of the list is high-quality from a precision standpoint but many similar documents have been ranked very low and therefore omitted from the

Source document	http://www.aabga.org
Source title	American Assoc. of Botanical Gardens and Arboreta
Source category	/home/gardens/clubs_and_associations

Settings: window size = 32, stem, dist and term weighting Sibling-Γ = **0.53**		
Rank	Sim	Category
1	0.16	/home/gardens/clubs_and_associations
2	0.15	/home/gardens/clubs_and_associations
5	0.13	/home/gardens/clubs_and_associations
10	0.11	/home/gardens/plants
20	0.10	/home/gardens/clubs_and_associations
50	0.07	/home/gardens/plants
100	0.06	/home/apartment_living/gardening

Settings: window size = 0, no stem, no term weighting Sibling-Γ = **0.30**		
Rank	Sim	Category
1	0.17	/reference/libraries/independent_libraries
2	0.15	/home/gardens/clubs_and_associations
5	0.14	business/industries/construction_and_maintenance
10	0.14	/business/industries/agriculture_and_forestry
20	0.13	/recreation/travel/reservations
50	0.13	/recreation/travel/reservations
100	0.13	business/industries/construction_and_maintenance

Figure 2.2: Similarity orderings obtained from two different strategies with respect to the same source document. We computed the sibling-Γ statistic (with respect the the ODP-familial ordering) for each ranked listing. For each document shown, we give the rank, the similarity to the source document according to the strategy, and the category (we omit the URL of the document).

top of the list. For our experiments, we generally concentrated on the sibling-Γ measure, as we are most interested in the fine-grained similarity distinctions that the various candidate strategies are able to make. As a preview of our results, and to illustrate the use of the sibling-Γ measure as an indicator of the relative quality of similarity-search strategies, we give an example of two strategies that yielded ranked listings with different sibling-Γ values in Figure 2.2.

2.3 Document Representation

In this section we will discuss the specific document representation and term weighting options we chose to evaluate using the technique outlined above. Let the Web

document u be represented by a bag

$$B_u = \{(w_u^1, f_u^1), \ldots, (w_u^k, f_u^k)\}$$

where w_u^i are terms used in representing u (e.g., terms found in the body of page u), and f_u^i are corresponding weights. In this section, we discuss different choices for which words to include in a document's bag (and with what weight); in Section 2.5 we will see how these different strategies performed under our evaluation criteria.

2.3.1 Choosing Terms

The following are 3 general approaches we consider for selecting the terms to include in the multiset representing a Web page u:

1. Words appearing in the body of u (a content-based approach). This approach exploits the intuition that similar pages contain many of the same words.

2. Identifiers (e.g. URLs) for each page v that links to u (a link-based approach). This approach exploits the intuition that similar documents are often cocited by other pages.

3. Words appearing inside or near an anchor in page v, when the anchor links to page u (an anchor-based approach). This approach exploits the intuition that similar pages are often described using the same words in anchors of pages that refer to them.

Content-based approaches ignore the available hyperlink data and are susceptible to spam. In particular, they rely solely on the information provided by the page's author, ignoring the opinions of the authors of other Web pages [8]. Link-based approaches, investigated in [21], suffer from the shortcoming that pages with few inlinks will not have sufficient citation data to measure similarity. This problem is especially pronounced when attempting to discover similarity relations for new pages that have not yet been cited (i.e., linked to) a sufficient number of times. As we will see in Section 2.5, under link-based approaches, the multisets for *most* documents (even related ones) are in fact disjoint.

The third type of approach, which relies on text in and near hyperlinks, referred to as the *anchor-window* [13], appears most useful for the Web similarity-search task. Indeed, the use of anchor-windows has been previously considered for a variety of other Web information retrieval tasks [2, 4, 13, 20]. The anchor-window can be thought of as a compact summary of the target page written by the author of the source page [2]. We expect that when aggregating the counts of terms from *all* anchor-windows that refer to a Web page, the frequency of relevant terms will dominate the frequency of irrelevant ones. Thus, the resulting distribution of term frequencies in the multiset is expected to be a signature that is a reliable, concise representation of the document.

These three general approaches can also be combined; we considered strategies where terms of two or all three types were included in the bag representations of pages. There are some additional details involved with each of these approaches that we describe next. For both the content and anchor-based approaches, we chose to remove all HTML comments, Javascript code, tags (except '`alt`' text), and non-alphabetic characters. For the anchor-based approach, we must also decide how many words to the left and right of an anchor A_{vu} (the anchor linking from page v to page u) should be included in B_u. We experimented with three strategies for this decision. In all cases, the anchor-text itself of A_{vu} is included, as well as the title of document u. The three windowing strategies are described next:

Basic: We choose some fixed window size W, and always include W words to the left, and W words to the right, of A_{vu}.[4] Specifically, we use $W \in \{0, 4, 8, 16, 32\}$.

Syntactic: We use sentence, paragraph, and HTML-region-detection techniques to dynamically bound the region around A_{vu} that gets included in B_u. The primary document features that are capable of triggering a window cut-off are paragraph boundaries, table cell boundaries, list item boundaries, and hard breaks which follow sentence boundaries. This technique resulted in very narrow windows that averaged close to only 3 words in either direction.

Topical: We use a simple technique for guessing topic boundaries at which to bound the region that gets included. The primary features that trigger this bounding

[4]Stopwords do not get counted when determining the window cutoff.

are heading beginnings, list ends, and table ends. A particularly common case handled by these windows was that of documents composed of several regions, each beginning with a descriptive header and consisting of a list of URLs on the topic of that header. Regions found by the TOPICAL heuristics averaged about 21 words in size to either side of the anchor.

2.3.2 Stemming and Stopwording

Once we have chosen the kinds of terms to include in a bag, we have the choice of modifying or filtering the terms somehow. One commonly used information retrieval technique is *stemming*, which removes suffixes so that word variants map to a single term. For instance, stemming maps the word "cars" to "car", since from an information retrieval perspective, the two are almost identical. Another common technique is to ignore *stopwords*, which are words such as "the" and "a" that have very little use from an information retrieval perspective. High frequency terms that appear in many documents are often candidates for stopword lists. In particular, we considered the use of a list containing roughly 800 stopwords constructed from high frequency terms. We explored the effect of three different stemming and stopwording variations:

NOSTEM: The term is left as is. If it appears in the stopword list, it is dropped.

STEM: The term is stemmed using Porter's well known stemming algorithm [65] to remove word endings. If the stemmed version of the term appears in the stemmed version of our stopword list, it is dropped.

STOPSTEM: The term is stemmed as above, for the purposes of checking whether the term stem is in the stopword list. If the stem is a stopword, the term is dropped, otherwise the original *unstemmed* term is added to the bag.

The STOPSTEM variation provides valuable insight into the effects of stemming, which are twofold: *(i)* Terms which should be collapsed, for instance 'computer' and 'computers,' get mapped to the same stem, 'computer' and *(ii)* many terms useless for detecting similarity which do not themselves appear frequently enough to get flagged as stopwords, e.g., 'informs,' will be flagged when using STOPSTEM, since the stem

'inform' is shared with the stopword 'information.' As we will see in Section 2.5, both effects manifest themselves to differing degrees in our experiments.

2.3.3 Term Weighting

A further consideration in generating document bags is how a term's frequency should be scaled. A standard technique employed in the information retrieval community involves using one of the variants of the term-frequency, inverse-document frequency (TF.IDF) weighting method [69]. A clear benefit of the TF.IDF family of weighting functions is that they attenuate the weight of terms with high document frequency (i.e., terms that appear in many documents). These monotonic term weighting schemes, however, amplify the weight of terms with very low document frequency. This amplification is in fact good for keyword-search queries, where a rare term in the query should be given the most importance. In the case where we are judging document similarities, however, rare terms are much less useful as they are often typos, rare names, or other nontopical terms that adversely affect the similarity measure. For example, if a query page contains the name of the author, who has a rare first name, we do not in general want to return a list of other documents whose authors have that first name. Therefore, we experimented with what we call a *nonmonotonic document frequency (NMDF)* term weighting scheme, since it attenuates both high *and* low document-frequency terms. The idea that mid-frequency terms have the greatest "resolving power" is not new [53, 69].

Another component of term weighting that we consider for anchor-text based strategies, which has a substantial impact on our quality metric, is *distance weighting*. Distance weighting is a scheme in which for a given anchor-window size, instead of treating all terms near a link A_{vu} equally, we weight them based on their distance from the link (with anchor-words themselves given distance 0). As we will see in Section 2.5, the use of a distance-based attenuation function that gives more weight to words closer to the link, in conjunction with large anchor-windows, significantly improves results under our evaluation measure.

The specifics of the weighting schemes we used and their relative performance are

described in detail in Section 2.5.2. Note that the weighting schemes we consider can produce nonintegral weights, whereas the multiplicity of terms in multisets are integers by definition. As a practical matter, we avoid this issue by scaling term weights by some factor (we used 1,000), and rounding to the nearest integer, although it only becomes necessary in the implementation of the min-hashing scheme (see Section 2.6).

2.4 Measure of Overlap

In the previous section we explained how we represent Web documents using bags (i.e., multisets). We now describe the metric we use to measure the overlap between two bags. The candidate similarity-search strategies we consider use the overlap of the bags for two pages, according to this metric, as their estimate for the similarity between the pages. Our metric is a variant of the *Jaccard coefficient*. The Jaccard coefficient of two *sets* A and B is defined as

$$sim_J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

We extend Jaccard from sets to bags by applying bag union and bag intersection in the above equation. In particular, we take the max and min multiplicity of terms, for the union and intersection operations, respectively.[5]

We focus on the Jaccard measure rather than the classical *cosine* measure because of a scalability consideration. For scaling our similarity-search strategies to massive Web-page repositories, we rely on the *Min-Hashing* technique [42]. The main idea here is to hash the bags for Web pages in a way such that bags are mapped to the *same* bucket with a probability equal to the Jaccard similarity between the bags. Creating such a hash function for the cosine measure is to our knowledge an open problem. On the other hand, creating such hashes is possible for the Jaccard measure (see [10, 42]). Note that the two measures are not significantly different from one

[5] As described in the previous section, by scaling our term weights and rounding, we ensure that terms have only integral multiplicities.

another; we verified that they did not lead to meaningful differences in our similarity rankings.

2.5 Experimental Results of Strategy Evaluation

For evaluating the various strategies discussed in Section 2.3, we employ the methodology described in Section 2.2. We sampled the Open Directory [60] to get 300 pairs of clusters from the third level in the hierarchy.[6] This set of pairs consisted of 100 pairs of sibling clusters, 100 pairs of cousin clusters, and 100 pairs of clusters with no familial relation. There were 144,767 URLs present in this test set of clusters. As our source of Web data, we used a crawl from the Stanford WebBase from January 2000, containing 42 million pages [41]. 51,469 of the URLs in the test clusters were linked to by some document in our crawl, and could thus be used by our anchor-based approaches. These test-set URLs were linked to by close to 1 million pages in our repository, all of which were used to support the anchor based strategy we studied.[7] This section describes the evaluation of the strategies suggested in Section 2.3.

We verified that all three of our Γ measures yield, with very few exceptions, the same relative order in performance of the candidate strategies. In a sense, this agreement is an indication of the robustness of our Γ measures. Here we report the results only for the sibling-Γ statistic — the graphs for the cousin-Γ and unrelated-Γ measures are similar. For some of the graphs shown in this section the difference of Γ scores between different strategies might seem quite small, i.e., second decimal digit. Notice, however, that in each graph we explore the effect of each parameter independently; when we add up the effect of all parameters, the difference becomes substantial.

[6] Any URLs present below the third level were collapsed into their third level ancestor category, as we described in Section 2.2.1.

[7] ODP pages themselves were excluded from the data set to avoid bias.

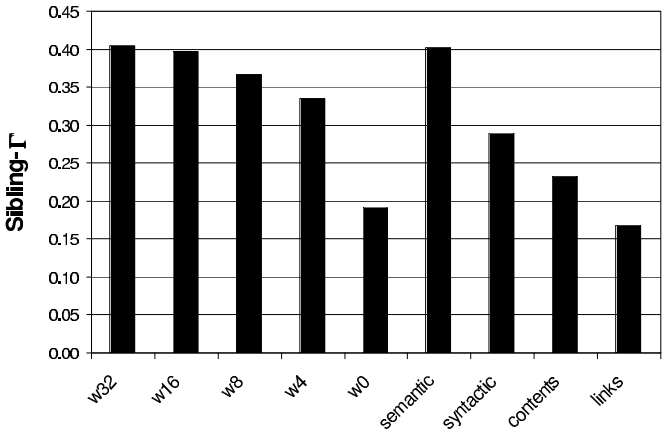


Figure 2.3: Document representations. Larger fixed anchor windows always gave better results, but topical dynamic windows achieved similar results with shorter average window size.

2.5.1 Results: Choosing Terms

In Figure 2.3, we show the sibling-Γ values for strategies for which bags are generated using various anchor-window sizes, using TOPICAL and SYNTACTIC window bounding, using purely links, and using purely page contents.

The anchor-based approach using the largest windows provides the best result according to our evaluation criteria. This result may seem counterintuitive; by taking small windows around inbound anchors, we would expect fewer spurious words to be present in a document's bag, providing a more concise representation. Further experiments revealed why larger windows provide benefit. Figure 2.4 shows the fraction of document pairs within the *same* Open Directory cluster that are *orthogonal* (i.e., have disjoint term bags) under a given representation. We see that with smaller window sizes, many documents that should be considered similar are orthogonal. In this case, no amount of reweighting or scaling can improve results; the representations simply do not provide enough accessible similarity information about these orthogonal pairs. We also see that, under the content and link approaches, documents in the same cluster are largely orthogonal. Under the link-based approach, *most* of the documents within a cluster are pairwise orthogonal. Inbound links are opaque descriptors. If two

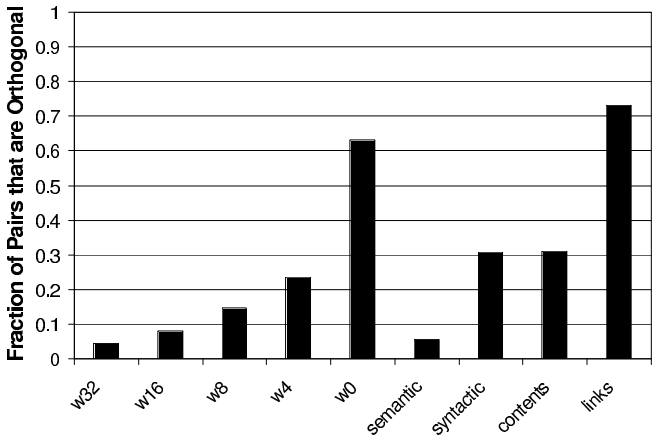


Figure 2.4: Intracluster orthogonality for the different strategies. wX refers to the anchor-window approach with a window size of X. Using small anchor-windows or using purely links resulted in document bags which were largely orthogonal, making similarities appear to be 0.

pages have many inlinks, but the intersection of their inlinks is empty, we can say very little about these two pages. It may be that they discuss the same topic, but perhaps because they are new, they are never cocited. In the case of the anchor-window-based approach, the chance that the bags for the two pages are orthogonal is much lower. Each inlink, instead of being represented by a single opaque URL, is represented by the descriptive terms that are the constituents of the inlink. Note that the pure link based approach shown is very similar to the *Cocitation Algorithm* of [21].[8]

We also experimented with dynamically sized SYNTACTIC and TOPICAL anchor-windows, as described in Section 2.3. These window types behave roughly according to their average window size, both in Γ values and orthogonality. Surprisingly, although the dynamic-window heuristics appeared to be effective in isolating the desired regions, any increase in region quality was overwhelmed by the trend of larger windows providing better results.

In addition to varying window size, we can also choose to include terms of multiple

[8]Furthermore we verified that the *Cocitation Algorithm* as described in [21] yields similar Γ scores to the scores for the 'links' strategy shown above.

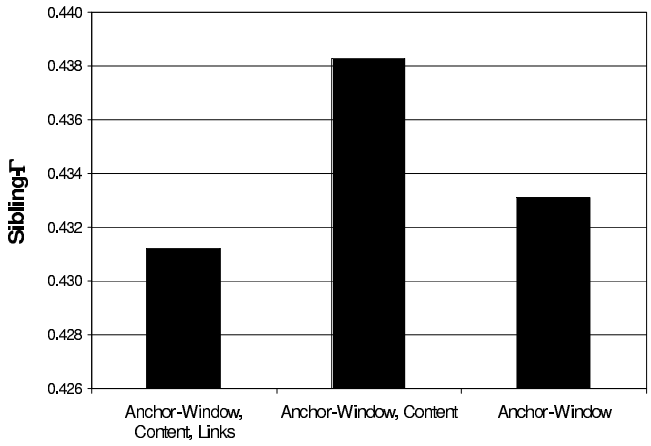


Figure 2.5: Three hybrid bag types. Adding page contents gave better results than using anchor-windows alone, though adding link identifiers lowered Γ.

types (anchor, content, or links, as described in Section 2.3) in our document representation. Figure 2.5 shows that by combining content and anchor-based bags, we can improve the sibling-Γ score.[9] The intuition for this variation is that if a particular document has very few inbound links, then the document's contents will dominate the bags. Otherwise, if the document has many inbound links, the anchor-window-based terms will dominate. In this way, the document's bag of terms will implicitly depend on as much information as is available. Note, however, that explicitly adding inlink URLs degrades performance, because the anchor-window approach subsumes any information that opaque inlink URLs can provide. With large anchor-windows, if two pages are in fact cocited often, they will share many of the same descriptive terms, due to overlapping anchor-windows. By adding inlink URLs, we end up reducing the bag overlap of truly similar pages.

2.5.2 Results: Term Weighting

In the previous section, we saw that the anchor-based approach with large windows performs the best. We can improve performance substantially under our evaluation

[9]All values in Figure 2.5 were generated with the distance-based term weighting scheme to be described.

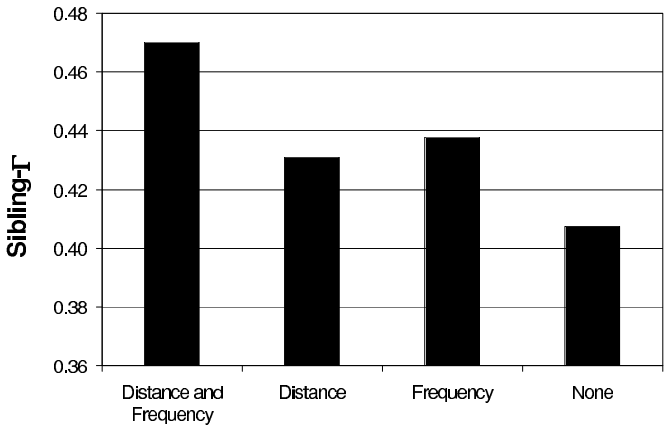


Figure 2.6: Term weighting: Frequency and distance weighting each improved results, and further improved results when combined.

criteria by weighting terms based on their distance from the anchor. We prevent ourselves from falling into the trap of making similar documents appear orthogonal (downside of small windows), while at the same time, not giving spurious terms too much weight (downside of large windows). Figure 2.6 shows the results when term weights are scaled by $\log_2(\frac{32}{1+\text{distance}(t,A_{vu})})$.

The results for frequency based weighting, shown in Figure 2.7, suggest that attenuating terms with low document frequency, in addition to attenuating terms with high document frequency (as is usually done), can increase performance. Let tf be a term's frequency in the bag, and *df* be the term's overall document frequency. Then in Figure 2.7, *log* refers to weighting with $\frac{tf}{1+\log_2(df)}$. *sqrt* refers to weighting with $\frac{tf}{\sqrt{df}}$. *NMDF* refers to weighting with the log-scale gaussian $tf \times e^{-\frac{1}{2}(\frac{\log(df)-\mu}{\sigma})^2}$ (see Figure 2.8).

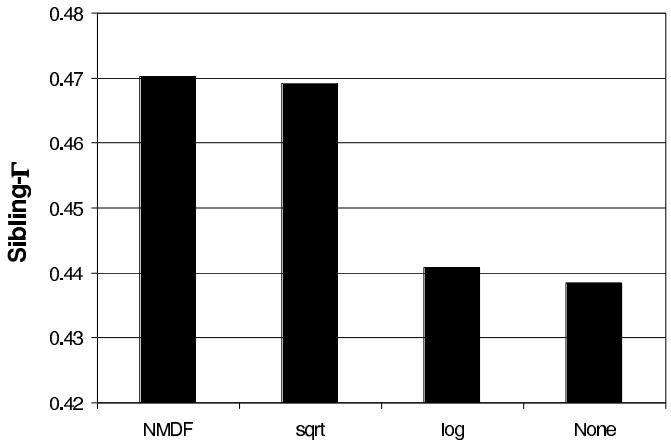


Figure 2.7: Types of frequency weighting: sqrt gave the best results of the monotonic frequency weighting schemes; NMDF gave slightly better results.

2.5.3 Results: Stemming and Stopwording

We now investigate the effects of our three stemming and stopwording approaches. Figure 2.9 shows the sibling-Γ values for the NOSTEM, STOPSTEM, and STEM strategies. We see that STOPSTEM improves the Γ value, and that STEM provides an additional (although much less statistically significant[10]) improvement. As mentioned in Section 2.3.2, the effect of STOPSTEM over NOSTEM is to increase the effective reach of the stopword list. Words that are not themselves detected as stopwords, yet share a stem with another word that was detected as a stopword, will be removed. The small additional impact of STEM over STOPSTEM is due to collapsing word variants into a single term.

2.6 Scaling to Large Repositories

The results of the previous section allowed us to choose the strategy that agrees most closely with the similarity judgements embodied in the ODP — namely, using

[10]The NOSTEM − STOPSTEM and STEM − STOPSTEM average differences are of the same approximate magnitude, however the pairwise variance of the STEM-STOPTEM is extremely high in comparison to the other pairwise variances.

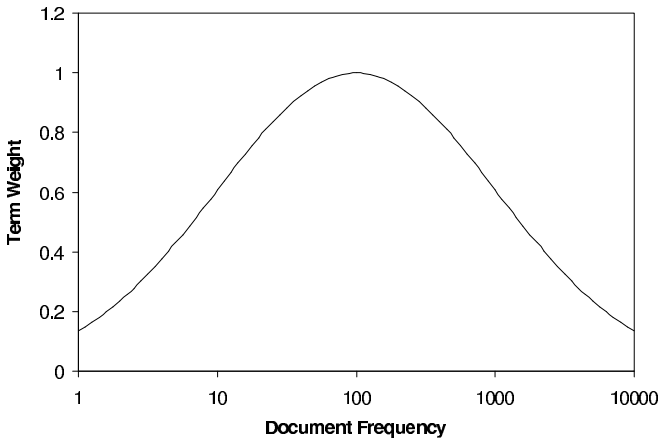


Figure 2.8: Nonmonotonic document frequency (NMDF) weighting.

size-32 anchor windows with page contents, in conjunction with stemming, distance weighting, and NMDF term weighting. In this section, we discuss how to scale this strategy to allow similarity search over large Web repositories. We begin with a definition and formal statement of the problem.

Definition 4 *We say two documents are* α-similar *if the Jaccard coefficient of their bags is greater than* α*.*

Problem 2 SimilarDocument **(efficiency considerations)***: Preprocess a repository of the Web* $\mathcal{W}$ *so that for each query Web-page* q *in* $\mathcal{W}$ *all Web pages in* $\mathcal{W}$ *that are* α*-similar to* q *can be found efficiently, for some fixed* α*.*

In this section, we develop a scalable algorithm, called IndexAllSimilar to solve the above problem for a realistic Web repository size.

In tackling Problem 2, there is a tradeoff between the work required during the preprocessing stage and the work required at query time to find the documents α-similar to q. For instance, we might imagine precomputing all Web-page similarities offline, and generating a similarity index that explicitly contained for each page, a list of all similar Web pages (an approach we considered in [38]). Although the query-time cost of this approach is low, a disadvantage is that when given a query document that was not available at indexing time, we can not return any results. In this section

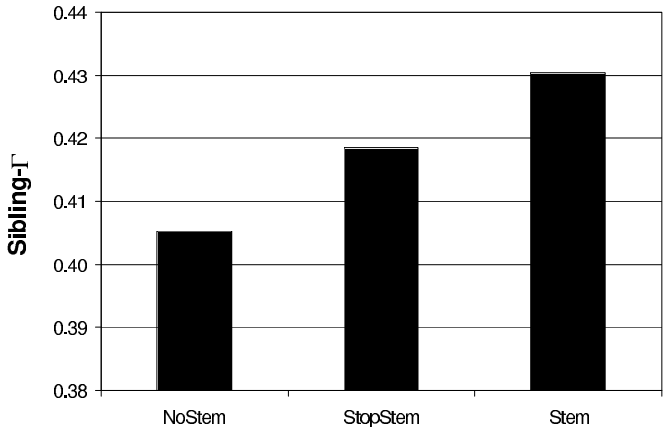


Figure 2.9: Stemming Variants: stemming gave the best results.

we discuss in detail an alternate approach that builds a specialized index during preprocessing that allows us to perform the similarity computation at query time. As we will describe, the index is compact and can be generated efficiently, allowing us to scale to large repositories with modest hardware resources. The computation required at query time is reasonable, and furthermore, given a query page that was not available to us at indexing time, we can use any partially available information (such as the title or contents of the query page, which the system can fetch in real-time) to allow the retrieval of similar pages.

A schematic view of the IndexAllSimilar algorithm is shown in Figure 2.10. In the next two sections, we explain IndexAllSimilar as a two stage algorithm. In the first stage we generate bags for each Web page in the repository. In the second stage, we generate a vector of signatures, known as Min-hash signatures, for each bag, and index these signature vectors to allow efficient retrieval both of document ids given signatures, and the signatures given document ids.

2.6.1 Bag Generation

As we explained in the previous sections, the bag for a document contains words (*i*) from the content of the document and (*ii*) from anchor-windows of other documents that link to it. Our bag generation algorithm scans through the Web repository

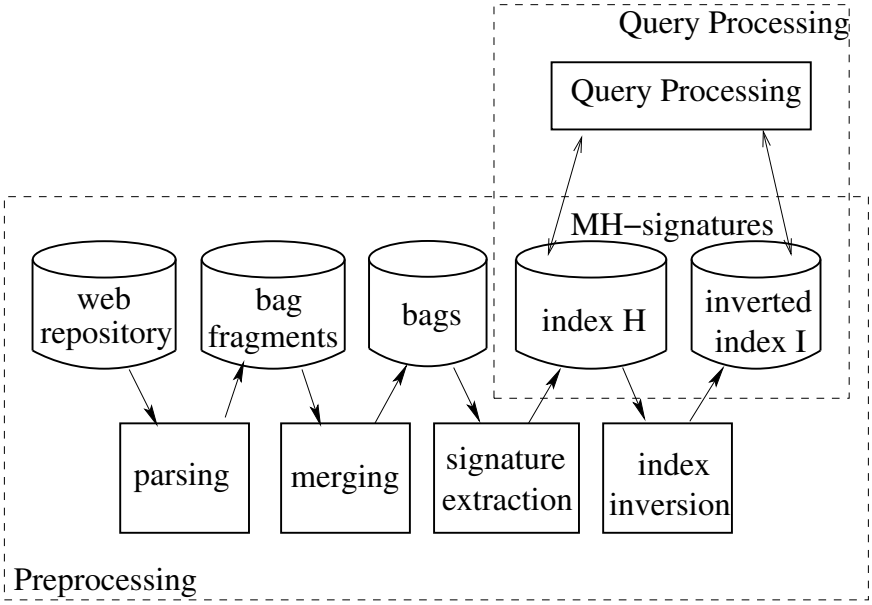


Figure 2.10: Schematic view of our approach.

and produces *bag fragments* for each document. A bag fragment for page u refers to the partial bag of terms that we have seen for u at some point during the scan — for instance, every time we encounter a link to u, we produce a bag fragment containing terms from that one anchor-window. For each document there is at most one *content bag fragment* (i.e., a partial bag containing terms from just the page's body) and possibly many *anchor bag fragments*. When generating bag fragments, we apply stemming and stopwording, and in the case of anchor bag fragments, distance weighting. After all bag fragments are generated, we must combine them using a sort-based merge operation to form the full bags for the pages, apply our NMDF scaling as discussed in Section 2.3.3, and normalize the weights to sum to a constant. Also, we ensure that our weights are integral by scaling by 1,000 and rounding to the nearest integer, so that our bag overlap measure is well defined.

2.6.2 Generation of the Document Similarity Index

At this point, we now have for each Web page, a representation as a bag of words $B = \{(w_1, f_1), \ldots, (w_k, f_k)\}$, where w are the words found in the content and anchor text of the document, and f are the corresponding integral weights. We describe next our indexing scheme.

There exists a family $\mathcal{H}$ of hash functions (see [12]) such that for each pair of documents u, v we have $Pr[h(u) = h(v)] = sim_J(u, v)$, where the hash function h is chosen at random from the family $\mathcal{H}$ and $sim_J(u, v)$ is the Jaccard similarity between the two pages' bags. The family $\mathcal{H}$ is defined by imposing a random order on the set of all words we have seen (known as the *lexicon*) and then representing each bag u by the lowest rank (according to that random order) element from B_u. For instance, let us assume that the only words we have seen in any bag are {*apple, grape, pear*}. Now consider a particular hash function h from the family $\mathcal{H}$. Say for instance the random order imposed on the lexicon for h is (*pear, apple, grape*). If the bag for u is {*grape, apple*}, then $h(u)$ is *apple*, since *apple* appeares earlier in the imposed lexicon ordering for h than *grape*.

We have to make a slight adjustment to this scheme to account for the fact that we are dealing with *multisets* rather than sets — we must treat each occurrence of a word in the multiset as a distinct term. In particular, we suffix the nth occurrence of a word in a multiset by the token *_n*. For instance, given the multiset {(*apple*, 2), (*pear*, 1)}, we would convert it to the set {*apple_1, apple_2, pear_1*}. Note that the Jaccard similarities of sets generated in this way are exactly the same as the bag-variant of the Jaccard measure applied to the original multisets. We modify the lexicon over which we generate random orders when computing signatures to consist of these suffixed terms.

In practice, it is quite inefficient to generate a fully random permutation of all words in the lexicon. Therefore, Broder et al. [12] use a family of random linear functions of the form $h(x) = ax + b \bmod p$. We use the same approach (see Broder et al. [11] and Indyk [42] for the theoretical background of this technique). Also, although a signature for a bag is conceptually a word (namely the word with the lowest rank according to a given random permutation of the lexicon), in practice we

Algorithm: ProcessQuery
Input: Query document q
Output: Similar documents
Let $mh_q = H[q]$ /* Fetch the MH-vector for q */
For each j from 1 to m /* Iterate over mh_q */
  /* For documents with the same j'th MH-signature as q */
  For each $doc_u \in I[j][mh_q[j]]$
    $sim[doc_u] + +$
Sort the set of docids $\{doc_i\}$ by their sim scores $sim[doc_i]$
Output $\{[doc_i, sim[doc_i]] | \frac{sim[doc_i]}{m} > \alpha\}$

Figure 2.11: Query Processing.

use a 4-byte numeric word identifier as the signature, in place of the word itself.

Based on the above property of the family $\mathcal{H}$ of hash functions, we compute for each bag a *vector* of *Min-hash signatures (MH-signatures)* that can be used to estimate the similarities of the corresponding bags. In particular, if we generate a vector mh_u of m MH-signatures for each document u, the expected fraction of the positions in which two pages share the same MH-signatures is equal to the Jaccard similarity of the bags for those two pages.

We now describe how we generate and index these signature vectors. We generate two data structures on disk. The first, H, consecutively stores mh_u for each document u (i.e., the m 4-byte MH-signatures for each document). Since our document ids are consecutively assigned, fetching these signatures for any document, given the document id, requires exactly 1 disk seek to the appropriate offset in H, followed by a sequential read of m 4-byte signatures. The second structure, I, is generated by inverting the first. For each position j in an MH-vector, and each MH-signature h that appears in position j in some MH-vector, $I[j][h]$ is a list containing id's for every document u such that the $mh_u[j] = h$. The algorithm for retrieving the ranked list of documents α-similar to the query document q, using the indexes H and I, is given in Figure 2.11.

When constructing the indexes H and I, the choice of m needed to ensure that documents that are α-similar to the query document are retrieved by ProcessQuery

depends solely on α; in particular, it is shown in [12] that the choice of m is independent of the number of documents, as well as the size of the lexicon. Since we found that documents within an Open Directory category have similarity of at least 0.15 to one another, we chose $\alpha = 0.15$. We can safely choose $m = 80$ for this value of α [18].[11]

2.7 Scalability Experiments

We employed the strategies that produced the best Γ values (see Section 2.5) in conjunction with the scalable algorithm we described above (see Section 2.6) to run an experiment on a sizable Web repository. In particular we used size-32 anchor-windows with distance and NMDF term weighting, stemming, and with content terms included. We provide a description of our dataset and the behavior of our algorithms, as well as a few examples from the results we obtained.

2.7.1 Efficiency Results

Our dataset was the January 2001 Stanford WebBase repository that contained roughly 120 million pages. For our large scale experiment, we used a 45 million page subset, which generated bags for 75 million pages.[12] After merging all bag fragments, we generated 80 MH-signatures ($m = 80$), each 4 bytes long, for each of the 75 million pages.

Three machines, each with a single AMD-K6 550MHz processor, were used to process the Web repository in parallel to produce the bag fragments. The subsequent steps (merging of fragments, MH-signature generation, and query processing) took place on a dual Pentium-III 933 MHz machine with 2 GB of main memory. The timing results of the various stages and index sizes are given in Figure 2.12. The

[11] We chose α and m heuristically; the properties of the Web as a whole differ from those of Open Directory. Given additional resources, decreasing α and increasing m would be appropriate.

[12] This latter number includes bags for pages that were linked to by pages in the 45 million Web page subset, but might not have been in the subset itself.

Algorithm step	Time
Generation of bag fragments	24 hours
Merging of anchor-bag fragments	8 hours
MH-signature generation	22 hours
Query Processing	< 3 seconds

Type of data	Space
Web repository (45M pages,compressed)	100 GB
Merged bags	42 GB
MH-signatures (H)	24 GB
Inverted MH-signatures (filtered) (I)	5 GB

Figure 2.12: Timing results and space usage for similarity index.

query processing step is dominated by the cost of accessing I, the smaller of the on-disk indexes. To improve performance, we filtered I to remove URLs of low indegree (3 or fewer inlinks). Note that these URLs remain in H, so that all URLs can appear as queries; some simply will not appear in *results*. Of course at a slight increase in query time (or given more resources), I need not be filtered in this way. Also note that if I is maintained wholly in main-memory (by partitioning it across several machines, for instance), the query processing time drops to a fraction of a second.

2.7.2 Quality of Retrieved Documents

Accurate comparisons with existing search engines are difficult, since one needs to make sure both systems use the same Web document collection. We have found however, that the "Related Pages" functionality of commercial search engines often return *navigationally*, as opposed to *topically*, similar results. For instance, `www.msn.com` is by in some sense similar to `moneycentral.msn.com`, as they are both part of Microsoft MSN. However, the former would probably not be a very useful result for someone looking for other financial sites. The use of our evaluation methodology has led us to strategies that reflect the topical notion of "similarity" embodied in the Open Directory. For illustration, we have provided some sample queries in Figure 2.13. In Figure 2.14 we have given the top 8 words in the bags for these query URLs.[13]

[13]For clarity, the terms displayed in Figure 2.13 were unstemmed with the most commonly occurring variant of the word.

MSN Money **moneycentral.msn.com**	**Weather.com** **www.weather.com**
MSN Money www.moneycentral.com	CNN.com - Weather www.cnn.com/WEATHER
Money Magazine www.pathfinder.com/money	Welcome to the Weather Underground www.princeton.edu/Webweather/ww.html
Welcome to Moneyextra www.moneyworld.co.uk	Rain or Shine www.rainorshine.com
Money www.money.com	UM Weather cirrus.sprl.umich.edu/wxnet
ETrade www.etrade.com	Weather for Active Lives www.intellicast.com/weather/
Money Club www.moneyclub.com	WeatherPost www.weatherpost.com
Morningstar - ... successful investing www.morningstar.net	Full-service weather company www.wni.com
The Money Page – ... Guide to Investment www.moneypage.com	Welcome to The Weather Underground www.wunderground.com

CNN Money **www.cnnfn.com**	**MP3.com: free mp3 downloads...** **www.mp3.com**
Financial markets, commodities, news www.bloomberg.com	International Music Network - About Us imnworld.com/about.html
Investors Business Daily www.investors.com	EMusic — World's Most Popular MP3 Service! www.emusic.com
Welcome to the new Barron's online www.barrons.com	CMJ: New Music First www.mp3now.com
Financial Times www.usa.ft.com	EMusic — World's Most Popular MP3 Service! www.goodnoise.com
CNN Money cnnfn.cnn.com	Lycos Music — Downloads mp3.lycos.com
CNBC on MSN Money Wizard www.cnbc.com	Audiogalaxy www.audiogalaxy.com
Financial Information Link Library www.mbnet.mb.ca/~russell	Listen www.listen.com
Wallstreet Journal Home Page update.wsj.com	LAUNCH.com - Discover New Music... www.launch.com

The Source for Java(TM) Technology **java.sun.com**	**CD Now** **www.cdnow.com**
The Source for Java(TM) Technology www.javasoft.com	CD Universe - Your Online Music Store www.cduniverse.com
developerWorks: Java technology www.ibm.com/java	The Orchard - ... music, artists, bands www.theorchard.com
The IT Industry Portal www.gamelan.com	Columbia House — Home Page www.columbiahouse.com
DevEdge Online - JavaScript Developer Central developer.netscape.com/tech/javascript/	Every CD www.everycd.com
Microsoft Visual J++ Home Page www.microsoft.com/visualj	CDconnection.com www.cdconnection.com
JavaScript World – Welcome! www.jsworld.com	Music Boulevard www.musicblvd.com
Java Boutique www.j-g.com/java	Music: CDs, records and tapes, oh my! www.gemm.com
JavaWorld.com www.javaworld.com	CD World cdworld.com

Figure 2.13: Sample queries and results.

URL	Top Terms in Bag (Desc. Order by Weight)
moneycentral.msn.com	money, finance, msn, website, moneycentral, stock, employment, microsoft
www.weather.com	weather, channel, forecasts, fbc, enter, travel, seek, best
www.cnnfn.com	finance, business, cnn, cnnfn, stock, market, street, money
www.mp3.com	music, audio, player, artist, napster, radio, band, million
java.sun.com	java, jdk, technology, microsystems, api, applet, spacer, platform
www.cdnow.com	music, cdnow, amazon, records, books, sports, best, entertainment

Figure 2.14: Top 8 words from sample bags.

2.8 Related Work

Most relevant to our work are algorithms for the "Related Pages" functionality provided by several major search engines. However, the details of these algorithms are not publicly available. Dean and Henzinger [21] propose algorithms, which we discussed in Sections 3.1 and 2.5.1, for finding related pages based on the link connectivity of the Web only and *not* on the text of pages. The idea of using anchor-text for document representation has been exploited in the past to attack a variety of information retrieval problems [2, 8, 13, 14, 20, 48]. Approaches algorithmically related to the ones presented in Section 2.6 have been used in [9, 12], although for the different problem of identifying mirror pages.

Part II

Topic Sensitive Search

Chapter 3

Topic-Sensitive Search

3.1 Introduction[1]

We discussed in the previous chapter how to discover pages similar to the page the user is currently browsing. We now turn to the question of how to exploit contextual information when ranking keyword-search results. Our system performs a series of offline link-analysis computations on the Web graph to generate a set of ranking vectors. Then at query time, our system analyzes the query and any available context, and calculates a final ranking for the query using these ranking vectors. We begin with a brief background on link-analysis algorithms, and proceed with a detailed discussion of our system.

Various link-based ranking strategies have been developed for improving Web-search query results. The HITS algorithm proposed by Kleinberg [48] relies on query-time processing to deduce the *hubs* (which are pages that link to "good" pages) and *authorities* (which are "good" pages) that exist in a subgraph of the Web consisting of both the results to a query and the local link-neighborhood of these results. The query time cost of HITS is nontrivial, as it requires performing iterative link-analysis computations after the query has been issued by the user.

The PageRank algorithm, introduced by Page et al. [61], precomputes a rank

[1]This chapter covers work we first presented in [31, 35, 37]. Portions reprinted, with permission, from "Topic-Sensitive PageRank: A Context-Sensitive Ranking Algorithm for Web Search," *IEEE Transactions on Knowledge and Data Engineering*, July/August 2003. ©2003 IEEE.

vector that provides a-priori "importance" estimates for all of the pages on the Web. In the case of PageRank, "importance" signifies a kind of link-popularity, or how often a page is linked to by other popular pages. The notion of importance captured by PageRank has a very precise definition that we discuss in Section 3.1.1. The traditional PageRank vector is computed once, offline, and is independent of the search query. At query time, these importance scores are used in conjunction with query-specific information retrieval (IR) scores, such as term occurrence frequencies, to rank the query results [8]. PageRank has a clear efficiency advantage over the HITS algorithm, as the query-time cost of incorporating the *precomputed* PageRank importance score for a page is low. Furthermore, as PageRank is generated using the entire Web graph, rather than a small subset, it is less susceptible to localized link spam. Figure 3.1 illustrates a system utilizing the standard PageRank scheme.

We propose an approach that (as with HITS) allows query-time information to influence the link-based score, yet (as with PageRank) requires minimal query-time processing. In our model, we compute offline a *set* of topic-sensitive PageRank vectors, each biased to a different topic, to create for each page a *set* of importance scores with respect to particular topics [35]. The biasing process involves the introduction of artificial links into the Web graph during the offline rank computation, and is described further in Section 3.1.1.

Chakrabarti et al. [15] and Pennock et al. [64] demonstrate that the properties of the Web graph are sensitive to page topic. In particular, it was found that pages tend to point to other pages that are on the same "broad" topic [64]. Although this property helps explain why a query-independent PageRank score can be useful for ranking, it also suggests that we may be able to improve the performance of link-based computations by taking into account page topics. By making PageRank topic-sensitive, we avoid the problem of heavily linked pages getting highly ranked for queries for which they have no particular authority [1]. Pages considered important in some subject domains may not be considered important in others, regardless of what keywords may appear either in the page or in anchor text referring to the page.

In this chapter, we consider two scenarios for enhancing keyword-search result rankings. In the first, we assume a user with a specific information need issues a

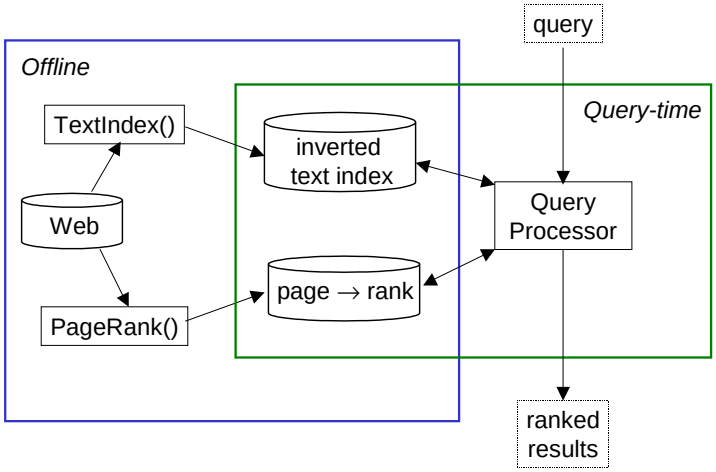


Figure 3.1: Simplified diagram illustrating a simple search engine utilizing the standard PageRank scheme.

query to our search engine by entering a query into a search box. In this scenario, we determine the topics most closely associated with the query, and use the appropriate topic-sensitive PageRank vectors for ranking the documents satisfying the query. This use of topical PageRank vectors ensures that the "importance" scores reflect a preference for the link structure of pages that have some bearing on the query. As with ordinary PageRank, the topic-sensitive PageRank score can be used as part of a scoring function that takes into account other IR-based scores. In the second scenario, we assume that in addition to the query, additional contextual information available. For instance, consider the case where a user who is viewing some Web page selects a phrase from the document for which he would like more information. In addition to the highlighted phrase, the surrounding terms and the Web page as a whole form a context for search. As an example, if a query for "architecture" was performed by highlighting a term in a Web page discussing famous building architects, we would like different results than if the query "architecture" was performed by highlighting a term in a page on CPU design. By selecting the appropriate topic-sensitive PageRank vectors based on the context of the query, we seek to provide more accurate search results. The *history* of queries issued by the user also constitutes a form of query context. Yet another source of context are the user's bookmarks or favorite pages. These various sources of search context are discussed in Section 3.4.

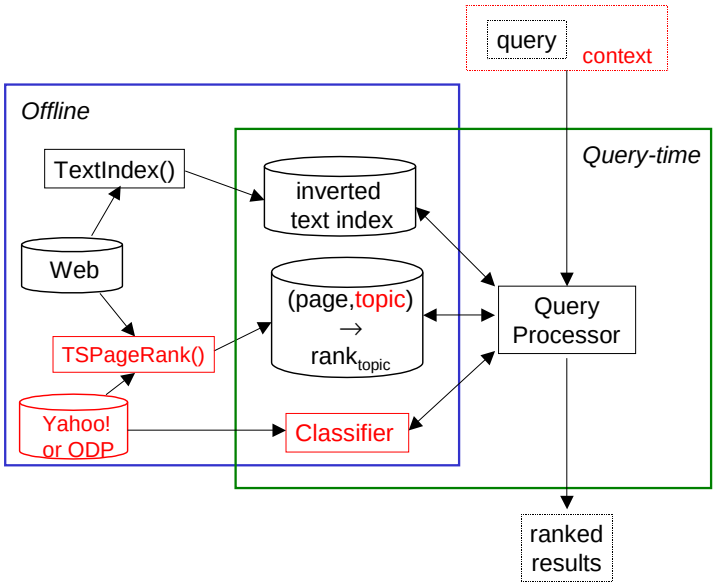


Figure 3.2: Illustration of our system utilizing topic-sensitive PageRank.

A summary of our approach follows. During the offline processing of the Web crawl, we generate 16 topic-sensitive PageRank vectors, each biased (as described in Section 3.1.1) using URLs from a top-level category from the Open Directory Project (ODP) [60]. At query time, we calculate the similarity of the query (and if available, the query or user context) to each of these topics. Then instead of using a single, global ranking vector, we take the linear combination of the topic-sensitive vectors, weighted using the similarities of the query (and any available context) to the topics. By using a *set* of topic-sensitive PageRank vectors, we determine more accurately which pages are truly the most important with respect to a particular query or query-context. Because the link-analysis computations are performed offline, during the preprocessing stage, the query-time costs are not much greater than that of the ordinary PageRank algorithm. An illustration of our topic-sensitive PageRank system is given in Figure 3.2.

3.1.1 Preliminaries

In this section we summarize the definition of PageRank [61] and review some of the mathematical tools we will use in analyzing and improving the standard iterative algorithm for computing PageRank.[2]

Underlying the definition of PageRank is the following basic assumption. A link from a Web page u to a Web page v can be viewed as evidence that v is an "important" page. In particular, the amount of importance conferred on v by u is proportional to the importance of u and inversely proportional to the number of pages u points to. Since the importance of u is itself not known, determining the importance for every page $i \in Web$ requires an iterative fixed-point computation.

We begin with an intuitive look at PageRank computation, and proceed with a more formal discussion later in the section. PageRank computation consists of a series of iterations that begins with an arbitrary assignment of ranks to pages, and then successively improves these ranks. Let $\vec{x}^{(k)}$ represent our estimate of PageRank at iteration k; i.e., let $x_i^{(k)}$ represents the rank for page i after iteration k. Since we do not have any way of knowing the true PageRank values prior to the computation, we start off with the uniform initial vector $\vec{x}^{(0)} = [1/n]_n$, where n is the number of pages in our Web graph. In other words, we begin by assigning every page the same rank (the exact value is irrelevant). This corresponds to the intuition that a-priori, we guess that all pages have the same importance. Then in each iteration, each page will distribute its ranks equally to its outlinks. Thus, pages with many inbound links will accumulate more rank. Note however that the amount of rank a page receives from an inbound link depends on the rank of the source of that link — this means that the PageRank computation will assign more rank to pages that have inbound links from pages with higher rank. This iterative process successively refines the assignment of ranks to pages, until the change in ranks between two iterations becomes smaller than some threshold. After this convergence is reached, the rank assigned to every page is almost exactly the same as the sum of the inbound rank from all of its parents. This captures the intuition that the rank of page should be based on the number as

[2]See Bianchini et al. [7] and Langville et al. [51] for a detailed analysis and summary of the existing research on PageRank.

inbound links it has as well as the ranks of the source of those links.

To allow for a more rigorous analysis of the necessary computation, we next describe an equivalent formulation of PageRank in terms of a random walk on the directed Web graph G. Let $u \rightarrow v$ denote the existence of an edge from u to v in G. Let $\deg(u)$ be the outdegree of page u in G. Consider a random surfer visiting page u at time k. In the next time step, the surfer chooses a node v_i from among u's out-neighbors $\{v|u \rightarrow v\}$ uniformly at random. In other words, at time $k+1$, the surfer lands at node $v_i \in \{v|u \rightarrow v\}$ with probability $1/\deg(u)$.

The PageRank of a page i is defined as the probability that at some particular time step $k > K$, the surfer is at page i. For sufficiently large K, and with minor modifications to the random walk, this probability is unique, illustrated as follows. Consider the Markov chain induced by the random walk on G, where the states are given by the nodes in G, and the stochastic transition matrix describing the transition from i to j is given by P with $P_{ij} = 1/\deg(i)$.

For P to be a valid transition probability matrix, every node must have at least 1 outgoing transition; e.g., P should have no rows consisting of all zeros. This property holds if G does not have any pages with outdegree 0, which does not hold for the Web graph. P can be converted into a valid transition matrix by adding a complete set of outgoing transitions to pages with outdegree 0. In other words, we can define the new matrix P' where all states have at least one outgoing transition in the following way. Let n be the number of nodes (pages) in the Web graph. Let $\vec{v}$ be the n-dimensional column vector representing a uniform probability distribution over all nodes:

$$\vec{v} = [\frac{1}{n}]_{n \times 1} \tag{3.1}$$

Let $\vec{d}$ be the n-dimensional column vector identifying the nodes with outdegree 0:

$$d_i = \begin{cases} 1 & \text{if } \deg(i) = 0, \\ 0 & \text{otherwise} \end{cases}$$

Then we construct P' as follows:

$$D = \vec{d} \cdot \vec{v}^T$$
$$P' = P + D$$

In terms of the random walk, the effect of D is to modify the transition probabilities so that a surfer visiting a dangling page (i.e., a page with no outlinks) randomly jumps to a page in the next time step, using the probability distribution given by $\vec{v}$.

By the Ergodic Theorem for Markov chains [29], the Markov chain defined by P' has a unique stationary probability distribution if P' is aperiodic and irreducible.[3] In the context of computing PageRank, the standard way of modifying the transition graph so that these properties are guaranteed to hold is to add a new set of complete outgoing transitions, with small transition probabilities, to *all* nodes, creating a complete transition graph. In matrix notation, we construct the irreducible, aperiodic Markov matrix P'' as follows:

$$E = [1]_{n \times 1} \times \vec{v}^T$$
$$P'' = cP' + (1 - c)E$$

In terms of the random walk, the effect of E is as follows. At each time step, with probability $(1-c)$, a surfer visiting any node will jump to a random Web page (rather than following an outlink). The destination of the random jump is chosen according to the probability distribution given in $\vec{v}$. We will refer to artificial jumps taken because of E as *teleportation*. The value of c generally used is in the literature is around 0.85, suggested in the original work on PageRank [61]. For convenience, we also use $\alpha = 1 - c$ later in the chapter to denote the probability of teleport. The PageRank vector $\vec{x}$ that we are interested in is precisely the stationary distribution of the Markov chain P'' for a given teleport constant c. The PageRank for page i is x_i.

[3]For a detailed explanation of these properties, we refer the reader to [44]. For the purposes of our discussion, these properties are equivalent to saying that the transition graph must be strongly connected, which is not the case for the Web graph without the modification that follows.

$\vec{y} = cP^T\vec{x}$;
$w = ||\vec{x}||_1 - ||\vec{y}||_1$;
$\vec{y} = \vec{y} + w\vec{v}$;

Algorithm 1: Computing $\vec{y} = A\vec{x}$

By redefining the vector $\vec{v}$ given in Equation 3.1 to be nonuniform, so that D and E add artificial transitions with nonuniform probabilities, the resultant PageRank vector can be biased to prefer certain kinds of pages. For this reason, we refer to $\vec{v}$ as the *personalization* vector. We denote to the stationary distribution of P'' for a given $\vec{v}$ (and some fixed c) as the personalized PageRank vector $\vec{x}(\vec{v})$.

For simplicity and consistency with prior work, the remainder of the discussion will be in terms of the transpose matrix, $A = (P'')^T$; i.e., the transition probability distribution for a surfer at node i is given by row i of P'', and column i of A. Note that the edges artificially introduced by D and E never need to be explicitly materialized, so this construction has no impact on efficiency or the sparsity of the matrices used in the computations. In particular, the matrix-vector multiplication $\vec{y} = A\vec{x}$ can be implemented efficiently using Algorithm 1.

Assuming that the probability distribution over the surfer's location at time 0 is given by $\vec{x}^{(0)}$, the probability distribution for the surfer's location at time k is given by $\vec{x}^{(k)} = A^k\vec{x}^{(0)}$. The unique stationary distribution of the Markov chain is defined as $\lim_{k\to\infty} x^{(k)}$, which is equivalent to $\lim_{k\to\infty} A^k x^{(0)}$, and is independent of the initial distribution $\vec{x}^{(0)}$. This stationary distribution is simply the principal eigenvector of the matrix $A = (P'')^T$, which is the PageRank vector we would like to compute.

The standard PageRank algorithm computes this principal eigenvector by starting with the uniform distribution $\vec{x}^{(0)} = \vec{v}$ and computing successive iterates $\vec{x}^{(k+1)} = A\vec{x}^{(k)}$ until convergence (i.e., it uses the *power method*). This algorithm is summarized in Algorithm 2. In Chapters 4 and 5, we will discuss the computation of PageRank in more detail, and introduce algorithms that improve upon the *power method* on large Web graphs.

function **pageRank**$(G, \vec{x}^{(0)}, \vec{v})$ {
Construct P from G: $P_{ji} = 1/\deg(j)$;
repeat
$\vec{x}^{(k+1)} = cP^T\vec{x}^{(k)}$;
$w = ||\vec{x}^{(k)}||_1 - ||\vec{x}^{(k+1)}||_1$;
$\vec{x}^{(k+1)} = \vec{x}^{(k+1)} + w\vec{v}$;
$\delta = ||\vec{x}^{(k+1)} - \vec{x}^{k}||_1$;
until $\delta < \epsilon$;
return $\vec{x}^{(k+1)}$;
}

Algorithm 2: PageRank

3.2 Topic-Sensitive PageRank

In our approach to topic-sensitive PageRank, we precompute the importance scores offline, as with ordinary PageRank. However, for each Web page, we compute an importance score *per topic*. At query time, these importance scores are combined based on the topics of the query and associated context to form a composite PageRank score for those pages matching the query. This score can be used in conjunction with other IR-based scoring schemes to produce a final rank for the result pages with respect to the query. As the scoring functions of commercial search engines are not known, in our work we do not focus on the effect of these IR scores (other than requiring that the query terms appear in the page).[4] We believe that the improvements to PageRank's precision will translate into improvements in overall search rankings, even after other IR-based scores are factored in. Note that the topic-sensitive PageRank score is more spam-resistant than other types of IR-based scores, making it desirable to depend on it more heavily.

3.2.1 ODP-biasing

The first step in our approach is to generate a set of biased PageRank vectors using a set of basis topics. This step is performed once, offline, during the preprocessing of

[4]For instance, most search engines use term weighting schemes which make special use of HTML tags.

the Web crawl. There are many possible sources for the basis set of topics. However, using a small basis set is important for keeping the preprocessing and query-time costs low. One option is to cluster the Web-page repository into a small number of clusters in the hopes of achieving a representative basis. We chose instead to use the freely available Open Directory Project (ODP) as a source of topics. In this chapter, we restrict our attention to the 16 top-level categories in the ODP hierarchy.

Let T_j be the set of URLs in the ODP category c_j. Then when computing the PageRank vector for topic c_j, in place of the uniform damping vector $\vec{v} = [\frac{1}{n}]_{n\times 1}$, we use the nonuniform vector $\vec{v_j}$ where

$$(v_j)_i = \begin{cases} \frac{1}{|T_j|} & i \in T_j, \\ 0 & i \notin T_j. \end{cases} \tag{3.2}$$

The PageRank vector for topic c_j is given by $\vec{x}(\vec{v_j})$. We also generate the single unbiased PageRank vector (denoted as NoBias) for the purpose of comparison. The choice of α will be discussed in Section 3.3.1.

We also compute the term occurrence matrix V, where V_{tj} is the total number of occurrences of term t in documents listed below class c_j of the ODP.

As mentioned previously, one could envision using other sources of topics; however, the ODP data is freely available, and as it is compiled by thousands of volunteer editors, is less susceptible to influence by any one party.

3.2.2 Query-Time Importance Score

The second step in our approach is performed at query time. Given a query q, let q' be the context of q. In other words, if the query was issued by highlighting the term q in some Web page u, then q' consists of the terms in u. We can also use only those terms in u nearby the highlighted term, as often times a single Web page may discuss a variety of topics. For ordinary queries not done in context, let $q' = q$. Using a multinomial naive-Bayes classifier [56], trained on the text of the pages in the ODP, we compute the class probabilities for each of the 16 top-level ODP classes, conditioned on q'. Let q'_i be the ith term in the query (or query context) q'. Then

given the query q, we compute for each c_j the following:

$$P(c_j|q') = \frac{P(c_j) \cdot P(q'|c_j)}{P(q')} \propto P(c_j) \cdot \prod_i P(q'_i|c_j) \tag{3.3}$$

$P(q'_i|c_j)$ is easily computed from the class term occurrence matrix V. In the absence of any information about the user, we make $P(c_j)$ uniform. For some particular user k, we can use a prior distribution $P_k(c_j)$ that reflects the interests of user k independent of any particular search query (e.g., by classifying his bookmarks).

Using a text index, we retrieve URLs for all documents containing the *original* query terms q. Finally, we compute the query-sensitive importance score of each of these retrieved URLs as follows. For convenience, let $\vec{r_j} = \vec{x}(\vec{v_j})$, so that $(r_j)_d$ is the rank of a search result d for the topic c_j. We compute the query-sensitive importance score s_{qd} for page d as follows.

$$s_{qd} = \sum_j P(c_j|q') \cdot (r_j)_d \tag{3.4}$$

The results are ranked according to this composite score s_{qd}, which can be used as component of a larger scoring function.

The above query-sensitive PageRank computation has the following probabilistic interpretation, in terms of the "random surfer" model [61]. Let w_j be the coefficient used to weight the jth rank vector, with $\sum_j w_j = 1$ (e.g., let $w_j = P(c_j|q)$). Then since PageRank is linear,[5] note that the following equality holds:

$$\sum_j [w_j \vec{x}(\vec{v_j})] = \vec{x}\big(\sum_j [w_j \vec{v_j}]\big) \tag{3.5}$$

holds. Thus we see that the following random walk on the Web yields the topic-sensitive score s_{qd}. With probability $1 - \alpha$, a random surfer on page u follows an outlink of u (where the particular outlink is chosen uniformly at random). With probability $\alpha P(c_j|q')$, the surfer instead jumps to one of the pages in T_j (where the

[5] See [45], or for a construction of the linear transformation from personalization vectors to personalized PageRank vectors, see Section 3.5.

particular page in T_j is chosen uniformly at random). The long term visit probability that the surfer is at page d is exactly given by the composite score s_{qd} defined above. Thus, topics exert influence over the final score in proportion to their affinity with the query (or query context).

3.3 Experimental Results

We conducted a series of experiments to measure the behavior of topic-sensitive PageRank. In Section 3.3.1 we describe the similarity measure we use to compare two rankings. In Section 3.3.2, we investigate how the induced rankings vary, based on both the topic used to bias the rank vectors as well as the choice of the bias factor α. In Section 3.3.3, we present results of a user study showing the retrieval performance of ordinary PageRank versus topic-sensitive PageRank. In Section 3.3.4, we describe how query context is utilized in our topic-sensitive PageRank scheme.

As a source of Web data, we used a Web crawl from the Stanford WebBase [41] performed in January 2001, containing roughly 120 million pages. Our crawl contained roughly 280,000 of the 3 million URLs in the ODP. For our experiments, we used 35 of the sample queries given in [23], which were in turn compiled from earlier papers.[6] The queries are listed in Table 3.1.

3.3.1 Similarity Measure for Induced Rankings

We use two measures when comparing rankings. The first measure, denoted $\mathrm{OSim}(\tau_1, \tau_2)$, indicates the degree of overlap between the top k URLs of two rankings, τ_1 and τ_2. We define the overlap of two sets A and B (each of size k) to be $\frac{|A \cap B|}{k}$. In our comparisons we will use $k = 20$. The overlap measure OSim gives an incomplete picture of the similarity of two rankings, as it does not indicate the degree to which the relative orderings of the top k URLs of two rankings are in agreement. Therefore, in addition to OSim, we use a second measure, KSim, based on Kendall's τ distance measure.[7]

[6] Several queries which produced very few hits on our repository were excluded.

[7] Note that the schemes for comparing top k lists recently proposed by Fagin et al. [24], also based on Kendall's τ distance measure, differ from KSim in the way normalization is done.

Table 3.1: Test queries used.

affirmative action	lipari
alcoholism	lyme disease
amusement parks	mutual funds
architecture	national parks
bicycling	parallel architecture
blues	recycling cans
cheese	rock climbing
citrus groves	san francisco
classical guitar	shakespeare
computer vision	stamp collecting
cruises	sushi
death valley	table tennis
field hockey	telecommuting
gardening	vintage cars
graphic design	volcano
gulf war	zen buddhism
hiv	zener
java	

For consistency with OSim, we will present our definition as a similarity (as opposed to distance) measure, so that values closer to 1 indicate closer agreement. Consider two partially ordered lists of URLs, τ_1 and τ_2, each of length k. Let U be the union of the URLs in τ_1 and τ_2. If δ_1 is $U - \tau_1$, then let τ_1' be the extension of τ_1, where τ_1' contains δ_1 appearing after all the URLs in τ_1.[8] We extend τ_2 analogously to yield τ_2'. We define our similarity measure KSim as follows:

$$\mathrm{KSim}(\tau_1, \tau_2) = \frac{|(u,v) : \tau_1', \tau_2' \text{ agree on order of } (u,v), u \neq v|}{(|U|)(|U|-1)} \tag{3.6}$$

In other words, $\mathrm{KSim}(\tau_1, \tau_2)$ is the probability that τ_1' and τ_2' agree on the relative ordering of a randomly selected pair of distinct URLs $(u, v) \in U \times U$.[9]

[8] The URLs in δ are placed with the *same* ordinal rank at the end of τ.

[9] A pair ordered in one list and tied in the other is considered a disagreement.

3.3.2 Effect of ODP-Biasing

In this section we measure the effects of topically biasing the PageRank computation. First, note that the choice of the bias factor α, discussed in Section 3.1.1, affects the degree to which the resultant vector is biased towards the topic vector used for $\vec{p}$. Consider the extreme cases. For $\alpha = 1$, the URLs in the bias set T_j will be assigned the score $\frac{1}{|T|}$, and all other URLs receive the score 0. Conversely, as α tends to 0, the content of T_j becomes irrelevant to the final score assignment.

We heuristically set $\alpha = 0.25$ after inspecting the rankings for several of the queries listed in Table 3.1. We did not concentrate on optimizing this parameter; although α affects the induced rankings of query results, the differences *across* different topically-biased PageRank vectors, for a fixed α, are much higher. For instance, for $\alpha = 0.05$ and $\alpha = 0.25$, we measured the average similarity of the induced rankings across our set of test queries, for each of our PageRank vectors.[10] The results are given in Table 3.2. We see that the average overlap between the top 20 results for the two values of α is high. Furthermore, the high values for KSim indicate high overlap as well as agreement (on average) on the relative ordering of these top 20 URLs for the two values of α. Chakrabarti et al. [15] suggest that the ideal choice of α may differ for different topics; choosing the optimal α for each topic is an avenue for future study. In the remainder of this chapter, experiments use $\alpha = 0.25$.

We now discuss the difference between rankings induced by different topically-biased PageRank vectors. We computed the average, across our test queries, of the pairwise similarity between the rankings induced by the different topically-biased vectors. The 5 most similar pairs, according to our OSim measure, are given in Table 3.3, showing that even the most similar topically-biased rankings have little overlap. Having established that the topic-specific PageRank vectors each rank the results substantially differently, we proceed to investigate which of these rankings is "best" for specific queries.

As an example, Table 3.4 shows the top 4 ranked URLs for the query "bicycling," using several of the topically-biased PageRank vectors. Note in particular that the

[10] We used 25 iterations of PageRank in all cases.

Table 3.2: Average similarity of rankings for $\boldsymbol{\alpha = 0.05}$ and $\boldsymbol{\alpha = 0.25}$.

Bias Set	$OSim$	$KSim$
NoBias	0.72	0.64
Arts	0.66	0.58
Business	0.63	0.54
Computers	0.70	0.60
Games	0.78	0.67
Health	0.73	0.62
Home	0.77	0.67
Kids & Teens	0.74	0.66
News	0.74	0.65
Recreation	0.62	0.55
Reference	0.68	0.57
Regional	0.60	0.52
Science	0.69	0.59
Shopping	0.66	0.55
Society	0.57	0.50
Sports	0.69	0.60
World	0.64	0.55

Table 3.3: Topic pairs yielding most similar rankings.

Bias-Topic Pair	$OSim$	$KSim$
(Games, Sports)	0.18	0.13
(NoBias, Regional)	0.18	0.12
(Kids & Teens, Society)	0.18	0.11
(Health, Home)	0.17	0.12
(Health, Kids & Teens)	0.17	0.11

ranking induced by the SPORTS-biased vector is of high quality. Also note that the ranking induced by the SHOPPING-biased vector leads to the high ranking of Websites selling bicycle-related accessories.

3.3.3 Query-Sensitive Scoring

In this section we consider the case where the query itself is used to generate the appropriate weights for topic-sensitive PageRank vectors when ranking keyword-search results. Given a query, our first task is to determine which of the rank vectors can best rank the results for the query. We found that using the quantity $P(c_j|q)$ as discussed in Section 3.2.2 yielded intuitive results for determining which topics are most closely associated with a query. In particular, for most of the test queries, the ODP categories with the highest values for $P(c_j|q)$ are intuitively the most relevant categories for the query. In Table 3.5, we list for several of the test queries the 3 categories with the highest values for $P(c_j|q)$. When computing the composite s_{qd} score in our experiments, we chose to use the weighted sum of only the rank vectors associated with the three topics with the highest values for $P(c_j|q)$, rather than all of the topics, to avoid introducing unnecessary noise.

To compare our query-sensitive approach to ordinary PageRank, we conducted a user study. We randomly selected 10 queries from our test set for the study, and found 5 volunteers. For each query, the volunteer was shown 2 result rankings; one consisted of the top 10 results satisfying the query, when these results were ranked with the unbiased PageRank vector, and the other consisted of the top 10 results for the query when the results were ranked with the composite s_{qd} score.[11] The volunteer was asked to select all URLs which were "relevant" to the query, in their opinion. In addition, they were asked to mark which of the two rankings was the better of the two, in their opinion. They were not told anything about how either of the rankings was generated.

Let a URL be considered *relevant* if at least 3 of the 5 volunteers selected it as relevant for the query. The *precision* then is the fraction of the top 10 URLs

[11]Both the title and URL were presented to the user. The title was a hyperlink to a current version of the Web page.

Table 3.4: Top results for the query "bicycling" when ranked using various topic-specific vectors.

NoBias	Arts
"RailRiders Adventure Clothing" www.RailRiders.com www.Waypoint.org/default.html www.Gorp.com/ www.FloridaCycling.com/	"Photo Contest & Gallery (Bicycling)" www.bikescape.com/photogallery/ www.trygve.com/ www.greenway.org/ www.jsc.nasa.gov/Bios/htmlbios/young.html
Business	**Computers**
"Recumbent Bikes and Kit Aircraft" www.rans.com www.BreakawayBooks.com java.oreilly.com/bite-size/ www.carbboom.com	"GPS Pilot" www.gpspilot.com www.wireless.gr/wireless-links.htm www.linkstosales.com www.LiftExperts.com/lifts.html
Games	**Kids and Teens**
"Definition Through Hobbies" www.flick.com/~gretchen/hobbies.html www.BellaOnline.com/sports/ www.npr.org/programs/wesun/puzzle/will.html www.trygve.com/	"Camp Shohola For Boys" www.shohola.com www.EarthForce.org www.WeissmanTours.com www.GrownupCamps.com/homepage.html
Recreation	**Science**
"Adventure travel" www.gorp.com/ www.GrownupCamps.com/homepage.html www.gorp.com/gorp/activity/main.htm www.outdoor-pursuits.org/	"Coast to Coast by Recumbent Bicycle" hypertextbook.com/bent/ www.SiestaSoftware.com/ www.BenWiens.com/benwiens.html www.SusanJeffers.com/jeffbio.htm
Shopping	**Sports**
"Cycling Clothing & Accessories for Women" www.TeamEstrogen.com/ www.ShopOutdoors.com/ www.jub.com.au/books/ www.bike.com/	"Swim, Bike, Run, & Multisport" www.multisports.com/ www.BikeRacing.com/ www.CycleCanada.com/ www.bikescape.com/photogallery/

Table 3.5: Estimates for $P(c_j|q)$ for a subset of the test queries.

alcoholism	
Health	0.47
Kids & Teens	0.20
Arts	0.06

bicycling	
Sports	0.52
Regional	0.13
Health	0.07

blues	
Arts	0.52
Shopping	0.12
News	0.08

citrus groves	
Shopping	0.34
Home	0.21
Regional	0.18

classical guitar	
Arts	0.75
Shopping	0.21
News	0.01

computer vision	
Computers	0.24
Business	0.14
Reference	0.09

cruises	
Recreation	0.65
Regional	0.18
Sports	0.04

death valley	
Regional	0.28
Society	0.14
News	0.10

field hockey	
Sports	0.89
Shopping	0.03
Reference	0.03

graphic design	
Computers	0.36
Business	0.23
Shopping	0.09

gulf war	
Society	0.21
Kids & Teens	0.18
Regional	0.17

hiv	
Health	0.40
News	0.19
Kids & Teens	0.14

java	
Computers	0.53
Games	0.10
Kids & Teens	0.06

lyme disease	
Health	0.96
Regional	0.01
Recreation	0.01

mutual funds	
Business	0.77
Regional	0.05
Home	0.05

parallel architecture	
Computers	0.70
Science	0.10
Reference	0.07

rock climbing	
Recreation	0.54
Regional	0.13
Sports	0.07

san francisco	
Sports	0.27
Regional	0.16
Recreation	0.10

shakespeare	
Arts	0.34
Reference	0.21
Kids & Teens	0.15

table tennis	
Sports	0.53
Shopping	0.14
Regional	0.09

telecommuting	
Business	0.70
Kids & Teens	0.04
Society	0.03

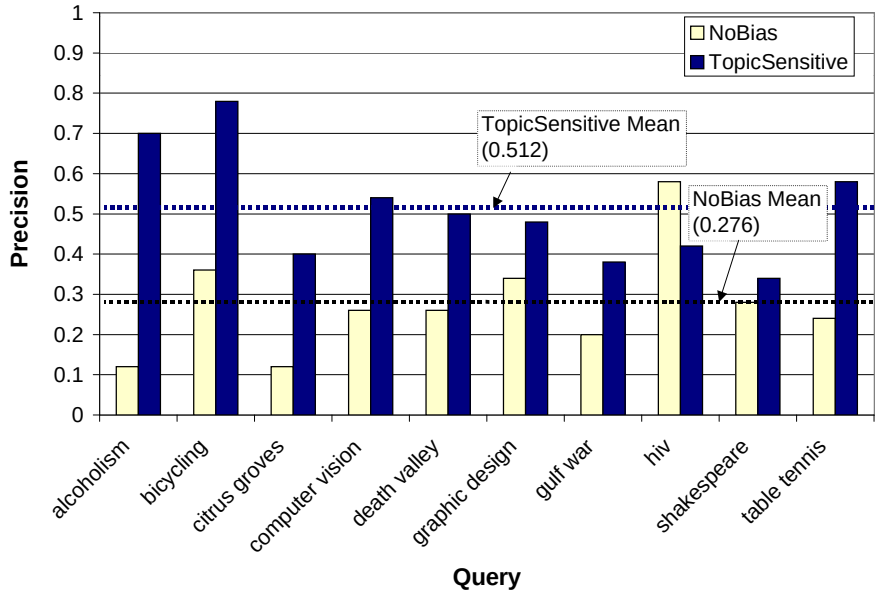


Figure 3.3: Precision @ 10 results for our test queries. The average precision over the ten queries is also shown.

that are deemed *relevant*. The precision of the two ranking techniques for each test query is shown in Figure 3.3. The average precision for the rankings induced by the topic-sensitive PageRank scores is substantially higher than that of the unbiased PageRank scores; 0.51 vs. 0.28. Furthermore, as shown in Table 3.6, for nearly all queries, a majority of the users selected the rankings induced by the topic-sensitive PageRank scores as the better of the two. These results suggest that the effectiveness of a query-result scoring function can be improved by the use of a topic-sensitive PageRank scheme in place of a generic PageRank scheme.

3.3.4 Context-Sensitive Scoring

In Section 3.3.3, weights for the topic-sensitive ranking vectors were chosen using the topics most strongly associated with the query term. If the search is done in context, for instance by highlighting a term in a Web page and invoking a search, then the context can be used instead of the query to determine the topics. Using the context can help disambiguate the query term and yield results that more closely reflect the intent of the user. We now illustrate how our system exploits query-context.

Table 3.6: Ranking scheme preferred by majority of users.

Query	Preferred by Majority
alcoholism	TopicSensitive
bicycling	TopicSensitive
citrus groves	TopicSensitive
computer vision	TopicSensitive
death valley	TopicSensitive
graphic design	TopicSensitive
gulf war	TopicSensitive
hiv	NoBias
shakespeare	Neither
table tennis	TopicSensitive

Consider the query "blues" taken from our test set. This term has several different senses; for instance it could refer to a musical genre, or to a form of depression. Two Web pages in which the term is used with these different senses, as well as short textual excerpts from the pages, are shown in Table 3.7. Consider the case where a user reading one of these two pages highlights the term "blues" to submit a search query. At query time, the first step of our system is to determine which topic best applies to the query in context. Thus, we calculate $P(c_j|q')$ as described in Section 3.2.2, using for q' the terms of the entire page. We can also include just a window of terms surrounding the highlighted query terms, using HTML markers to denote window boundaries. For the first page (discussing music), $\text{argmax}_{c_j} P(c_j|q')$ is Arts, and for the second page (discussing depression), $\text{argmax}_{c_j} P(c_j|q')$ is Health. The next step is to use a text index to fetch a list of URLs for all documents containing the term "blues" — the highlighted term for which the query was issued. Finally, the URLs are ranked using the appropriate ranking vector that was selected using the $P(c_j|q')$ values (i.e., either Arts or Health). Table 3.8 shows the top 5 URLs for the query "blues" using the topic-sensitive PageRank vectors for Arts, Health. We see that as desired, most of the results ranked using the Arts-biased vector are pages discussing music, while all of the top results ranked using the Health-biased vector discuss depression. The context of the query allows the system to pick the appropriate topic-sensitive ranking vector, and yields search results reflecting the appropriate sense of the search term.

Table 3.7: Two different search contexts for the query "blues."

That Blues Music Page	**Postpartum Depression & the 'Baby Blues'**
http://www.fred.net/turtle/blues.shtml	http://familydoctor.org/handouts/379.html
. . . If you're stuck for new material, visit Dan Bowden's Blues and Jazz Transcriptions - lots of older blues guitar transcriptions for you historic blues fans . . .	. . . If you're a new mother and have any of these symptoms, you have what is called the "baby blues." "The blues" are considered a normal part of early motherhood and usually go away within 10 days after delivery. However, some women have worse symptoms or symptoms last longer. This is called "postpartum depression." . . .

Table 3.8: Results for query "blues" using two different ranking vectors.

Arts
Britannica Online www.britannica.com
BandHunt.com Genres (Music) www.bandhunt.com/genres.html
Artist Information (Music) www.artistinformation.com/index.html
Billboard.com (Music charts) www.billboard.com
Soul Patrol (Music) www.soul-patrol.com

Health
Northern County Psychiatric Associates News www.baltimorepsych.com/news.htm
Seasonal Affective Disorder www.ncpamd.com/seasonal.htm
Women's Mental Health www.ncpamd.com/Women's_Mental_Health.htm
Wing of Madness Depression Support Group www.wingofmadness.com
Country Nurse Online www.countrynurse.com

3.4 Sources of Search Context

In addition to the example described above, there are a variety of sources of context that may be used with our scheme. For instance, the history of queries issued leading up to the current query is another form of query context. A search for "basketball" followed up with a search for "Jordan" presents an opportunity for disambiguating the latter. As another example, most modern search engines incorporate some sort of hierarchical directory, listing URLs for a small subset of the Web, as part of their search interface.[12] The current node in the hierarchy that the user is browsing at constitutes a source of query context. When browsing URLs at Top/Arts, for instance, any queries issued could have search results (from the entire Web index) ranked with the Arts rank vector, rather than either restricting results to URLs listed in that particular category, or not making use of the category whatsoever. In addition to these types of context associated directly with the query, we can also utilize user context, such as the user's bookmarks and recently visited Web pages. As mentioned in Section 3.2.2, we integrate user context by selecting a nonuniform prior, $P_k(c_j)$, based on how closely the user's context accords with each of the basis topics.

When attempting to utilize the aforementioned sources of search context, mediating the personalization of PageRank via a set of basis topics yields several benefits over attempting to explicitly construct a personalization vector.

Flexibility: For any kind of context, we can compute the context-sensitive PageRank score by using a classifier to compute the similarity of the context with the basis topics and then weighting the topic-sensitive PageRank vectors appropriately. We can treat such diverse sources of search context such as bookmarks, browsing history, and query history uniformly.

Transparency: The topically-biased rank vectors have intuitive interpretations. If we see that our system is giving undue preference to certain topics, we can tune the classifier used on the search context, or adjust topic weights

[12] See for instance `http://directory.google.com/Top/Arts/` or `http://dir.yahoo.com/Arts/`.

manually. When utilizing user context, the users themselves can be shown what topics the system believes represent their interests.

Privacy: Certain forms of search context raise potential privacy concerns. This concern could be alleviated with a *client-side* program that generates the user profile locally based on the user's context, and that sends only the summary information, consisting of the weights assigned to the basis topics, over to the server. When making use of query-context, if the user is browsing sensitive personal documents, they may be more comfortable if the search client sent to the server topic weights rather than the actual document text surrounding the highlighted query term.

Efficiency: For a small number of basis topics (such as the 16 ODP categories), both the query-time cost and the offline preprocessing cost of our approach is low, and practical to implement with current Web indexing infrastructure.

3.5 Comparison of Approaches to Personalizing PageRank

As observed in [45], PageRank is linear with respect to the personalization vector $\vec{v}$. In this section, we derive the linear transformation explicitly, and then discuss how several approaches to personalizing PageRank can all be characterized as ways of using a set of basis PageRank vectors to approximate arbitrary personalizations.

Let n be the number of pages on the Web. Recall that $\vec{x}(\vec{v})$ denotes the n-dimensional personalized PageRank vector corresponding to the n-dimensional personalization vector $\vec{v}$. Let $\vec{e}$ be the n-vector whose elements are all $e_i = 1$. $\vec{x}(\vec{v})$ can be computed by solving the following eigenvalue problem, where $A = cP^T + (1-c)\vec{v}\vec{e}^T$:

$$\vec{x} = A\vec{x} \tag{3.7}$$

Rewriting the above, we see that

$$\vec{x} = cP^T\vec{x} + (1-c)\vec{v} \tag{3.8}$$

$$\vec{x} - cP^T\vec{x} = (1-c)\vec{v} \tag{3.9}$$

$$(I - cP^T)\vec{x} = (1-c)\vec{v} \tag{3.10}$$

$I - cP$ is strictly diagonally dominant, so that $I - cP$ is invertible. Therefore, $(I - cP)^T = I - cP^T$ is also invertible. Thus, we get that

$$\vec{x} = (1-c)(I - cP^T)^{-1}\vec{v} \tag{3.11}$$

Let $Q = (1-c)(I-cP^T)^{-1}$. Thus, we see that Q is the linear transformation that gives us the PageRank vector $\vec{x}(\vec{v})$ given some personalization vector $\vec{v}$. By letting $\vec{v} = \vec{e}_i$, where $\vec{e}_i$ is the ith elementary vector[13] we see that the ith column of the matrix Q is $\vec{x}(\vec{e}_i)$, i.e., the personalized PageRank vector corresponding to the personalization vector $\vec{e}_i$.

The columns of Q comprise a complete basis for personalized PageRank vectors, as any personalized PageRank vector can be expressed as a convex combination[14] of the columns of Q. For any personalization vector $\vec{v}$, the corresponding personalized PageRank vector is given by $Q\vec{v}$. This formulation corresponds to the original approach to personalizing PageRank suggested by Page et al. [61] that allows for personalization on arbitrary sets of pages. Unfortunately, this first approach, which uses the complete basis for personalized PageRank, is infeasible in practice. Computing the dense matrix Q offline is impractical, as it amounts to running Algorithm 2 a total of n times, using every Web page in turn as the sole nonzero element in $\vec{v}$. Computing $\vec{x}(\vec{v})$ at query time using the Power Method is also impractical.

However, we can compute low-rank approximations of Q, denoted as $\hat{Q}$, that still allow us to achieve the benefit of personalized PageRank analysis. Rather than using a full basis (i.e., all the columns of Q), we can choose to use a reduced basis, e.g.,

[13] i.e., $\vec{e}_i$ has a 1 in the ith component, and zeros elsewhere

[14] A convex combination is a linear combination where the linear coefficients are nonnegative and sum to 1.

using only $k \leq n$ personalized PageRank vectors, each of which is a column (or more generally, a convex combination of the columns) of Q. In this case, we cannot express all personalized PageRank vectors, but only those corresponding to convex combinations of the PageRank vectors in the reduced basis set:

$$\vec{x}(\vec{w}) = \hat{Q}\vec{w} \tag{3.12}$$

where w is a stochastic k-vector representing weights over the k basis vectors.

The following three approaches each approximate Q by the matrix $\hat{Q}$, although they differ substantially in their computational requirements and in the granularity of personalization achieved.

Topic-Sensitive PageRank: The Topic-Sensitive PageRank scheme we discussed earlier in this chapter computes an $n \times k$ approximation to Q using k topics, e.g., the 16 top level topics of the Open Directory [60]. Column j of $\hat{Q}$ is given by $\vec{x}(\vec{v}_j)$, where $\vec{v}_j$ is a dense vector generated using a classifier for topic T_j; $(v_j)_i$ represents the (normalized) degree of membership of page i to topic j. Note that in this scheme, each column of $\hat{Q}$ must be generated independently, so that k must be kept fairly small (e.g., $k < 100$). This scheme uses a fairly coarse basis set limiting the degree to which it can personalize rankings to a specific individual. The use of a good set of representative basis topics ensures that the approximation $\hat{Q}$ will be useful. In Topic-Sensitive PageRank, $\hat{Q}$ is generated completely offline. Convex combinations are taken at query time, using the context of the query to compute the appropriate topic weights. In terms of the random-surfer model of PageRank, this scheme restricts the choice of teleportation transitions so that the random surfer can teleport to a topic T_j with some probability w_j, followed by a teleport to a particular page i with probability $(v_j)_i$.

Modular PageRank: The Modular PageRank approach proposed by Jeh and Widom [45] computes an $n \times k$ matrix using the k columns of Q corresponding to

highly ranked pages. In addition, that work provides an efficient scheme for computing these k vectors, in which *partial* vectors are computed offline and then composed at query time, making it feasible to have $k \geq 10^4$. In terms of the random-surfer model of PageRank, this scheme restricts the choice of teleportation transitions so that the random surfer can teleport to certain highly ranked pages, rather than to arbitrarily chosen sets of pages. A direct comparison of the relative granularity of this approach to the topic-sensitive approach is difficult. Although the basis set of personalized PageRank vectors is much larger in this scenario, they must come from personalization vectors $\vec{v}$ with singleton nonzero entries corresponding to highly ranked pages. However, the larger size of the basis set does allow for finer grained modulation of rankings.

BlockRank: The personalized BlockRank algorithm we describe in Section 5.4.4 of Chapter 5 computes an $n \times k$ matrix corresponding to k "blocks," where each block corresponds to a host, such as www-db.stanford.edu or nlp.stanford.edu. If we restrict personalizations to the block level, that work computes a matrix $\hat{Q}$ in which column j corresponds to $\vec{x}(\vec{v_j})$, where $\vec{v_j}$ represents the *local PageRank* of the pages in block j. The BlockRank algorithm is able to exploit the Web's inherent block structure to efficiently compute many of these block-oriented basis vectors, so that $k \geq 10^3$ is feasible. In terms of the random-surfer model of PageRank, this scheme restricts the choice of teleportation transitions so that the random surfer can teleport to block (i.e., host, rather than page) B_j with probability w_j, followed by a teleport to a particular page i in block B_j with probability $(v_j)_i$.

3.6 Related Work

Most relevant our work are the original algorithms for Web graph link-analysis, HITS and PageRank [48, 61]. Bharat and Henzinger [6] augment the HITS algorithm with content analysis to improve precision for the task of retrieving documents related to a query topic (as opposed to retrieving documents that exactly satisfy the user's information need). Chakrabarti et al. [13] make use of HITS for automatically compiling

resource lists for general topics. Zhang et al. [76] modifies the way teleports are done in the PageRank random walk to increase the spam-resistance of PageRank. Gyongyi et al. [30] suggest a PageRank-like computation to identify "trusted" (i.e., nonspam) pages.

The idea of personalizing the PageRank computation was suggested in the original work on PageRank [61], although in a form that was not practical on a large scale. Rafiei and Mendelzon [66] proposed using the set of Web pages that contain some term as a bias set for influencing the PageRank computation, with the goal of returning terms for which a given page has a high reputation. An approach for enhancing search rankings by generating a PageRank vector for each possible query term was proposed by Richardson and Domingos [68] with favorable results. That approach requires considerable processing time and storage, and is not easily extended to make use of user and query *context*. Diligenti et al. [22] propose topic-specific PageRank scores for enhancing *vertical search*, or search geared towards a particular niche of users. Our approach to biasing the PageRank computation is novel in its use of a small number of representative basis topics, in conjunction with a classifier, to achieve personalized, context-sensitive search.

Chapter 4

Computing PageRank by Power Extrapolation

4.1 Introduction[1]

Because our approach to context-sensitive search relies on computing many PageRank vectors, as described in the previous chapter, it is important to speed up the standard Power Method (presented earlier as Algorithm 2 in Section 3.1.1) used for computing PageRank. Recall that the Power Method is an iterative algorithm that begins with an initial vector $\vec{x}^{(0)}$ and successively refines it by repeatedly multiplying by a matrix corresponding to the Web link graph. In this chapter, we introduce a numerical technique that uses *extrapolation* to speed up the computation of PageRank vectors by reducing the number of iterations required. Extrapolation refers to the use of two or more vectors from different Power Method iterations (known as *iterates*) to compute a better approximation to the PageRank vector. More specifically, extrapolation techniques eliminate error along what are known as *nonprincipal eigenvectors* from the current iterate.[2] It is the error along these directions that limits the rate of convergence of the Power Method iterates to the final value (the *principal* eigenvector of the link matrix A). By eliminating them, we can speed up the convergence of the standard Power Method.

[1] This chapter covers work we first presented in [32, 40]

[2] For a detailed discussion on eigenvectors and eigenvalues, we refer the reader to [25].

A practical extrapolation method for accelerating the computation of PageRank, called Quadratic Extrapolation, was first presented by Kamvar et al. in [47]. That work assumed that none of the nonprincipal eigenvalues of the Web link matrix were known. By deriving the eigenvalues of the link matrix, we construct here a simpler and more effective extrapolation algorithm. We show empirically on an 80 million page Web crawl that this algorithm speeds up the computation of PageRank by 30% over the standard Power Method. The speedup in number of iterations is basically equivalent to that of Quadratic Extrapolation, but the method presented here is much simpler to implement, and has negligible overhead, so that its wallclock-speedup is higher by 9%.[3]

4.2 Experimental Setup

In the following sections, we will be introducing a series of algorithms for computing PageRank, and discussing the rate of convergence achieved on realistic datasets. Our experimental setup was as follows. We used a link graph, which we denote as LargeWeb, generated from a crawl of the Web done by the Stanford WebBase project in January 2001 [41]. LargeWeb contains roughly 80M nodes, with close to a billion links, and requires 3.6GB of storage. Dangling pages, which are pages without outlinks, were removed as described in [61]. The graph was stored using an adjacency list representation, with pages represented by 4-byte integer identifiers. On an AMD Athlon 1533MHz machine with a 2-way linear RAID disk volume and 3.5GB of main memory, each iteration of the Power Method (Algorithm 2 of Section 3.1.1) on the 80M page LargeWeb dataset takes roughly 10 minutes. Given that the full Web contains billions of pages, and computing PageRank generally requires roughly 50 applications of Algorithm 1, the need for fast methods is clear.

We measured the relative rates of convergence of the algorithms that follow using

[3]Note that because the Quadratic Extrapolation algorithm does not assume the second eigenvalue of the matrix is known, it is more widely applicable (outside of the context of PageRank) than the algorithms presented here.

the L_1 norm of the residual vector; i.e.,

$$||Ax^{(k)} - x^{(k)}||_1$$

We show in Appendix 4.A that the residual is a good proxy for measuring the convergence of the ranking of search results that are induced by successive iterates. In other words, in the case of PageRank, using the L_1 residual to measure how much the intermediate vector is changing in successive iterations of the Power Method accurately reflects the degree to which search results change when ranked by the successive iterates.

4.3 Power Method

We now take a more detailed look at the Power Method, to provide the background for our extrapolation methods.

4.3.1 Formulation

One way to compute the stationary distribution of a Markov chain is by explicitly computing the distribution at successive time steps, using $\vec{x}^{(k)} = A\vec{x}^{(k-1)}$, until the distribution converges. This method is called the Power Method for computing the principal eigenvector of A, and is given as Algorithm 3. The Power Method is the oldest method for computing the principal eigenvector of a matrix, and is at the heart of both the motivation and implementation of the original PageRank algorithm. The original PageRank algorithm is simply Algorithm 3 where Algorithm 1) is used to perform the matrix-vector multiplication $A\vec{x}$.

The intuition behind the convergence of the power method is as follows. Let us write our initial vector, $\vec{x}^{(0)}$, as a linear combination of the eigenvectors of A:[4]

$$\vec{x}^{(0)} = \vec{u}_1 + \alpha_2\vec{u}_2 + \ldots + \alpha_m\vec{u}_m \tag{4.1}$$

[4]For simplicity, we have assumed that the start vector $\vec{x}^{(0)}$ lies in the subspace spanned by the eigenvectors of A. Although this may not hold, it does not affect convergence guarantees.

function $\vec{x}^{(n)} = PowerMethod()$ {
$\vec{x}^{(0)} = \vec{v}$;
$k = 1$;
repeat
  $\vec{x}^{(k)} = A\vec{x}^{(k-1)}$;
  $\delta = ||x^{(k)} - x^{(k-1)}||_1$;
  $k = k + 1$;
until $\delta < \epsilon$;
}

Algorithm 3: Power Method

Recall that when we multiply an eigenvector u_k (with corresponding eigenvalue λ_k) of some matrix M by the matrix, we get the vector $\lambda_k u_k$. Furthermore, recall that the first eigenvalue of a Markov matrix (such as our Web link matrix A) is 1. Thus, we have

$$\vec{x}^{(1)} = A\vec{x}^{(0)} = \vec{u}_1 + \alpha_2\lambda_2\vec{u}_2 + \ldots + \alpha_m\lambda_m\vec{u}_m \tag{4.2}$$

and

$$\vec{x}^{(n)} = A^n\vec{x}^{(0)} = \vec{u}_1 + \alpha_2\lambda_2^n\vec{u}_2 + \ldots + \alpha_m\lambda_m^n\vec{u}_m \tag{4.3}$$

Since $\lambda_n \leq \ldots \leq \lambda_2 < 1$, we know that $A^{(n)}\vec{x}^{(0)}$ approaches $\vec{u}_1$ as n grows large. Therefore, we can see why the Power Method converges to the principal eigenvector of the Markov matrix A.

4.3.2 Operation Count

A single iteration of the Power Method consists of the single matrix-vector multiplication $A\vec{x}^{(k)}$. In general, such a multiplication is an an $O(n^2)$ operation (where the matrix is $n \times n$). However, if the multiplication is performed using Algorithm 1 of Section 3.1.1, which exploits the sparsity of the underlying link matrix, it is an $O(n)$ operation. In particular, the average outdegree of pages on the Web has been found to be around 7 [67]. On our datasets, we observed an average of around 8 outlinks per page.

4.3.3 Results and Discussion

If λ_2 is close to 1, then the power method is slow to converge, because n must be large before λ_2^n is close to 0. As we show in Section 4.4, the eigengap (i.e., the quantity $|\lambda_1| - |\lambda_2|$) for the Web Markov matrix A is given exactly by the teleport probability $1-c$. Thus, when the teleport probability is large, the Power Method works reasonably well. However, for a large teleport probability (and with a uniform personalization vector $\vec{v}$), the effect of certain kinds of link spam is increased, and pages can achieve unfairly high rankings. A high teleport probability means that every page is given a fixed "bonus" rank. Link spammers can make use of this bonus to generate structures to inflate the importance of certain pages. Low teleport probabilites can also increase the effect of spam – in that case, *rank sinks*, or sets of pages that do not link outside of their respective set, can have their ranks amplified. Those cases provide some intuition as to why c is usually kept around 0.85.

In Figure 4.1, we show the convergence on the LargeWeb dataset of the Power Method for $c \in \{0.80, 0.85, 0.90\}$ using a uniform damping vector $\vec{v}$. Note that increasing c slows down convergence. Since each iteration of the Power Method takes 10 minutes, computing 50 iterations requires over 8 hours. As the full Web is estimated to contain over eight billion static pages, using the Power Method on Web graphs close to the size of the Web would require several days of computation. In the next sections, we describe how to remove the error components of $x^{(k)}$ along the directions of certain nonprincipal eigenvectors, thus increasing the effectiveness of Power Method iterations.

4.4 The Second Eigenvalue of the PageRank Matrix

Before describing our algorithm for accelerating PageRank computations using extrapolation, we first analyze the properties of the nonprincipal eigenvectors of the Web graph. More specifically, we analytically derive the modulus of the second eigenvalue of the Web link matrix, as formally stated in the following theorem:[5]

[5]The proof that follows was derived jointly in [40].

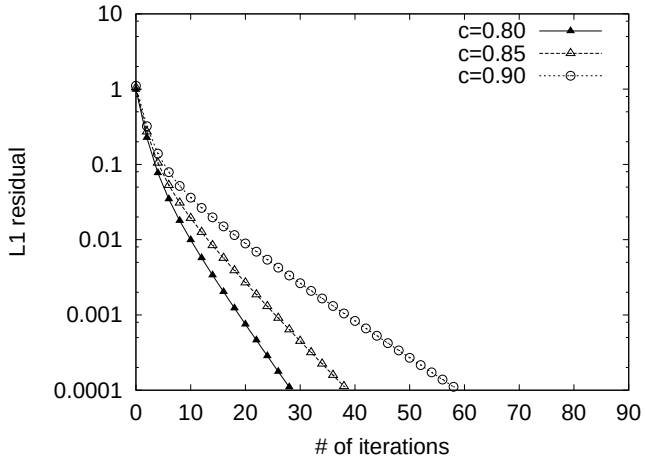


Figure 4.1: Comparison of convergence rate for the standard Power Method on the LargeWeb dataset for $c \in \{0.80, 0.85, 0.90\}$.

Theorem 1 *Let P be an $n \times n$ row-stochastic matrix. Let c be a real number such that $0 \leq c \leq 1$. Let E be the $n \times n$ rank-one row-stochastic matrix $E = \vec{e}\vec{v}^T$, where $\vec{e}$ is the n-vector whose elements are all $e_i = 1$, and $\vec{v}$ is an n-vector that represents a probability distribution.*[6]
Define the matrix $A = [cP + (1-c)E]^T$. Its second eigenvalue $|\lambda_2| \leq c$.

Theorem 2 *Further, if P has at least two irreducible closed subsets (which is the case for the Web link matrix), then the second eigenvalue of A is given by $\lambda_2 = c$.*

4.4.1 Notation and Preliminaries

P is an $n \times n$ row-stochastic matrix. E is the $n \times n$ rank-one row-stochastic matrix $E = \vec{e}\vec{v}^T$, where $\vec{e}$ is the n-vector whose elements are all $e_i = 1$ and $\vec{v}$ is an n-vector whose elements are all non-negative and sum to 1. A is the $n \times n$ column-stochastic matrix:

$$A = [cP + (1-c)E]^T \tag{4.4}$$

[6] I.e., a vector whose elements are nonnegative and whose L_1 norm is 1.

We denote the ith eigenvalue of A by λ_i, and the corresponding eigenvector by $\vec{x_i}$.

$$A\vec{x_i} = \lambda_i \vec{x_i} \tag{4.5}$$

By convention, we choose eigenvectors $\vec{x_i}$ such that $||\vec{x_i}||_1 = 1$. Since A is column-stochastic, $\lambda_1 = 1$, and $1 \geq |\lambda_2| \geq \ldots \geq |\lambda_n| \geq 0$.

We denote the ith eigenvalue of P^T as γ_i, and its corresponding eigenvector as $\vec{y_i}$: $P^T\vec{y_i} = \gamma_i\vec{y_i}$. Since P^T is column-stochastic, $\gamma_1 = 1, 1 \geq |\gamma_2| \geq \ldots \geq |\gamma_n| \geq 0$.

We denote the ith eigenvalue of E^T as μ_i, and its corresponding eigenvector as $\vec{z_i}$: $E^T\vec{z_i} = \mu_i\vec{z_i}$. Since E^T is rank-one and column-stochastic, $\mu_1 = 1, \mu_2 = \ldots = \mu_n = 0$.

An $n \times n$ row-stochastic matrix M can be viewed as the transition matrix for a Markov chain with n states.
For any row-stochastic matrix M, $M\vec{e} = \vec{e}$.
A set of states S is a *closed subset* of the Markov chain corresponding to M if and only if $i \in S$ and $j \notin S$ implies that $M_{ij} = 0$.
A set of states S is an *irreducible closed subset* of the Markov chain corresponding to M if and only if S is a closed subset, and no proper subset of S is a closed subset.
Intuitively speaking, each irreducible closed subset of a Markov chain corresponds to a leaf node in the strongly connected component (SCC) graph of the directed graph induced by the nonzero transitions in the chain.
Note that E, P, and A^T are row stochastic, and can thus be viewed as transition matrices of Markov chains.

4.4.2 Proof of Theorem 1

We first show that Theorem 1 is true for $c = 0$ and $c = 1$.

Case 1: $c = 0$
If $c = 0$, then, from Equation 4.4, $A = E^T$. Since E is a rank-one matrix, $\lambda_2 = 0$.

Thus, Theorem 1 is proved for c=0.

CASE 2: $c = 1$
If $c = 1$, then, from Equation 4.4, $A = P^T$. Since P^T is a column-stochastic matrix, $|\lambda_1| = 1$, so that $|\lambda_2| \leq 1$. Thus, Theorem 1 is proved for c=1.

CASE 3: $0 < c < 1$
We prove this case via a series of lemmas.

Lemma 1. The second eigenvalue of A has modulus $|\lambda_2| < 1$.

Proof. Consider the Markov chain corresponding to A^T. If the Markov chain corresponding to A^T has only one irreducible closed subchain S, and if S is aperiodic, then the chain corresponding to A^T must have a unique eigenvector with eigenvalue 1, by the Ergodic Theorem [29]. So we simply must show that the Markov chain corresponding to A^T has a single irreducible closed subchain S, and that this subchain is aperiodic. Lemma 1.1 shows that A^T has a single irreducible closed subchain S, and Lemma 1.2 shows this subchain is aperiodic.

Lemma 1.1 There exists a unique irreducible closed subset S of the Markov chain corresponding to A^T.
Proof. We split this proof into a proof of existence and a proof of uniqueness.
Existence. Let the set U be the states with nonzero components in $\vec{v}$. Let S consist of the set of all states reachable from U along nonzero transitions in the chain. S trivially forms a closed subset. Further, since every state has a transition to U, no subset of S can be closed. Therefore, S forms an irreducible closed subset.
Uniqueness. Every closed subset must contain U, and every closed subset containing U must contain S. Therefore, S must be the unique irreducible closed subset of the chain.

Lemma 1.2 The unique irreducible closed subset S is an aperiodic subchain.

Proof. From Theorem 5 in the appendix, all members in an irreducible closed subset have the same period. Therefore, if at least one state in S has a self-transition, then the subset S is aperiodic. Let u be any state in U. By construction, there exists a self-transition from u to itself. Therefore, S must be aperiodic.

From Lemmas 1.1 and 1.2, and the Ergodic Theorem, $|\lambda_2| < 1$ and Lemma 1 is proved.

Lemma 2. *The second eigenvector $\vec{x_2}$ of A is orthogonal to e: $\vec{e}^T\vec{x_2} = 0$.*

Proof. Since $|\lambda_2| < |\lambda_1|$ (by Lemma 1), the second eigenvector $\vec{x_2}$ of A is orthogonal to the first eigenvector of A^T by Theorem 3 in the appendix. From Section 4.4.1, the first eigenvector of A^T is $\vec{e}$. Therefore, x_2 is orthogonal to $\vec{e}$.

Lemma 3. $E^T\vec{x_2} = 0$

Proof. By definition, $E = \vec{e}\vec{v}^T$, and $E^T = \vec{v}\vec{e}^T$. Thus, $E^T\vec{x_2} = \vec{v}\vec{e}^T\vec{x_2}$. From Lemma 2, $\vec{e}^T\vec{x_2} = 0$. Therefore, $E^T\vec{x_2} = 0$.

Lemma 4. The second eigenvector $\vec{x_2}$ of A must be an eigenvector $\vec{y_i}$ of P^T, and the corresponding eigenvalue is $\gamma_i = \lambda_2/c$.

Proof. From Equation 4.4 and Equation 4.5:

$$cP^T\vec{x_2} + (1-c)E^T\vec{x_2} = \lambda_2\vec{x_2} \tag{4.6}$$

From Lemma 3 and Equation 4.6, we have:

$$cP^T\vec{x_2} = \lambda_2\vec{x_2} \tag{4.7}$$

We can divide through by c to get:

$$P^T \vec{x_2} = \frac{\lambda_2}{c} \vec{x_2} \tag{4.8}$$

If we let $\vec{y_i} = \vec{x_2}$ and $\gamma_i = \lambda_2 / c$, we can rewrite Equation 4.7.

$$P^T \vec{y_i} = \gamma_i \vec{y_i} \tag{4.9}$$

Therefore, $\vec{x_2}$ is also an eigenvector of P^T, and the relationship between the eigenvalues of A and P^T that correspond to $\vec{x_2}$ is given by:

$$\lambda_2 = c\gamma_i \tag{4.10}$$

Lemma 5. $|\lambda_2| \leq c$

Proof. We know from Lemma 4 that $\lambda_2 = c\gamma_i$. Because P is stochastic, we have that $|\gamma_i| \leq 1$. Therefore, $|\lambda_2| \leq c$, and Theorem 1 is proved.

4.4.3 Proof of Theorem 2

Recall that Theorem 2 states: If P has at least two irreducible closed subsets, $\lambda_2 = c$.

Proof.
Case 1: $c = 0$
This is proven in Case 1 of Section 4.4.2.

Case 2: $c = 1$
This is proven trivially from Theorem 3 in the appendix.

Case 3: $0 < c < 1$
We prove this case as follows. We assume P has at least two irreducible closed subsets.

We then construct a vector x_i that is an eigenvector of A and whose corresponding eigenvalue is $\lambda_i = c$. Therefore, $|\lambda_2| \geq c$, and there exists a $\lambda_i = c$. From Theorem 1, $\lambda_2 \leq c$. Therefore, if P has at least two irreducible closed subsets, then $\lambda_2 = c$.

Lemma 6. *Any eigenvector y_i of P^T that is orthogonal to e is an eigenvector x_i of A. The relationship between eigenvalues is $\lambda_i = c\gamma_i$.*

Proof. It is given that $\vec{e}^T\vec{y_i} = 0$. Therefore,

$$E^T\vec{y_i} = \vec{v}\vec{e}^T\vec{y_i} = 0 \tag{4.11}$$

By definition,

$$P^T\vec{y_i} = \gamma_i\vec{y_i} \tag{4.12}$$

Therefore, from Equations 4.4, 4.11, and 4.12,

$$A\vec{y_i} = cP^T\vec{y_i} + (1-c)E^T\vec{y_i} = cP^T\vec{y_i} = c\gamma_i\vec{y_i} \tag{4.13}$$

Therefore, $A\vec{y_i} = c\gamma_i\vec{y_i}$ and Lemma 6 is proved.

Lemma 7. There exists a $\lambda_i = c$.

Proof. We construct a vector $\vec{x_i}$ that is an eigenvector of P and is orthogonal to $\vec{e}$. From Theorem 3 in the appendix, the multiplicity of the eigenvalue 1 for P is equal to the number of irreducible closed subsets of P. Thus we can find two linearly independent eigenvectors $\vec{y_1}$ and $\vec{y_2}$ of P^T corresponding to the dominant eigenvalue 1. Let

$$k_1 = \vec{y_1}^T\vec{e} \tag{4.14}$$

$$k_2 = \vec{y_2}^T\vec{e} \tag{4.15}$$

If $k_1 = 0$, let $\vec{x_i} = \vec{y_1}$, else if $k_2 = 0$, let $\vec{x_i} = \vec{y_2}$. If $k_1, k_2 > 0$, then let $\vec{x_i} = \vec{y_1}/k_1 - \vec{y_2}/k_2$. Note that $\vec{x_i}$ is an eigenvector of P^T with eigenvalue exactly 1 and that x_i is orthogonal to e. From Lemma 6, $\vec{x_2}$ is an eigenvector of A corresponding to eigenvalue c. Therefore, the eigenvalue λ_i of A corresponding to eigenvector $\vec{x_i}$ is $\lambda_i = c$.

Therefore, $|\lambda_2| \geq c$, and there exists a $\lambda_i = c$. However, from Theorem 1, $\lambda_2 \leq c$. Therefore, $\lambda_2 = c$ and Theorem 2 is proved.[7]

4.5 Extrapolation Methods

Using the proof given in the previous section that the modulus of the second eigenvalue of A is given by the damping factor c, we derive here extrapolation methods that are easy to implement. We present a series of algorithms that exploit known eigenvalues of A to accelerate the Power Method for computing PageRank. In particular, one class of nonprincipal eigenvectors of A correspond to simple cycles of different lengths embedded in the Web. As a simple example, consider a Web graph with two pages that form a cycle. Assuming a random jump factor of $c = 0.8$, we have the following link matrix:

$$A = .8 \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + .2 \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.1 & 0.9 \\ 0.9 & 0.1 \end{pmatrix}$$

Notice that the vector

$$\vec{x} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

is an eigenvector of A with eigenvalue -0.8:

$$A\vec{x} = \begin{pmatrix} -0.8 \\ 0.8 \end{pmatrix} = -0.8\vec{x}$$

[7] As we will see in the next section, there may be additional eigenvalues with modulus c, such as $-c$.

Longer cycles lead to additional eigenvectors with modulus c, including those that include imaginary components and have imaginary eigenvalues (e.g., c, $-c$, ci, and $-ci$) [44]. We will see in Sections 4.5.2 and 4.5.3 how we can use extrapolation to eliminate these kinds of eigenvectors to speed up the convergence of the Power Method.

4.5.1 Simple Extrapolation

Formulation

The simplest extrapolation rule naively assumes that the iterate $\vec{x}^{(k-1)}$ can be expressed as a linear combination of the eigenvectors u_1 and u_2, where u_2 has eigenvalue c.

$$\vec{x}^{(k-1)} = \vec{u}_1 + \alpha_2 \vec{u}_2 \tag{4.16}$$

Now consider the current iterate $\vec{x}^{(k)}$; because the Power Method generates iterates by successive multiplication by A, we can write $\vec{x}^{(k)}$ as

$$\begin{aligned} \vec{x}^{(k)} &= A\vec{x}^{(k-1)} && (4.17)\\ &= A(\vec{u}_1 + \alpha_2 \vec{u}_2) && (4.18)\\ &= \vec{u}_1 + \alpha_2 \lambda_2 \vec{u}_2 && (4.19) \end{aligned}$$

Plugging in $\lambda_2 = c$, we see that

$$\vec{x}^{(k)} = \vec{u}_1 + \alpha_2 c \vec{u}_2 \tag{4.20}$$

The above allows us to solve for $\vec{u}_1$ in closed form:

$$\vec{u}_1 = \frac{\vec{x}^{(k)} - c\vec{x}^{(k-1)}}{1 - c} \tag{4.21}$$

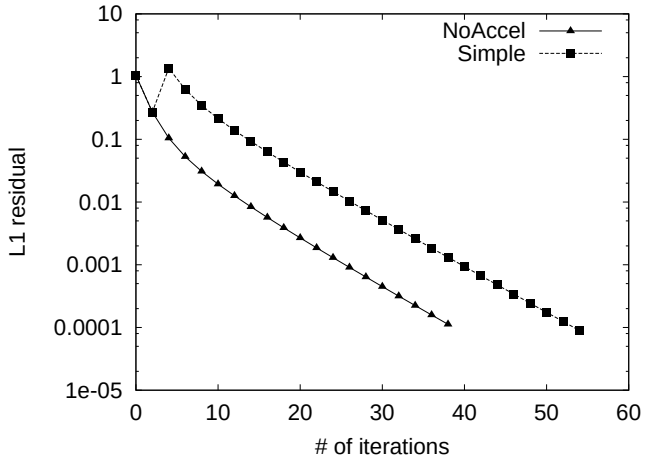


Figure 4.2: Comparison of convergence rates for Power Method and Simple Extrapolation on LargeWeb for $c = 0.85$. Notice that convergence is *slower*, making this an ineffective extrapolation technique.

Results and Discussion

Figure 4.2 shows the convergence of Simple Extrapolation and the standard Power Method, where there was one application of Simple Extrapolation at iteration 3 of the Power Method. Simple Extrapolation is not effective, as the assumption that c is the only eigenvalue of modulus c is inaccurate. In fact, by doubling the error in the eigenspace corresponding to eigenvalue $-c$, this extrapolation technique *slows down* the convergence of the Power Method.

4.5.2 A^2 Extrapolation

Formulation

The next extrapolation rule assumes that the iterate $\vec{x}^{(k-2)}$ can be expressed as a linear combination of the eigenvectors u_1, u_2, and u_3, where u_2 has eigenvalue c and u_3 has eigenvalue $-c$.

$$\vec{x}^{(k-2)} = \vec{u}_1 + \alpha_2\vec{u}_2 + \alpha_3\vec{u}_3 \tag{4.22}$$

Now consider the current iterate $\vec{x}^{(k)}$; because the Power Method generates iterates

by successive multiplication by A, we can write $\vec{x}^{(k)}$ as

$$\vec{x}^{(k)} = A^2\vec{x}^{(k-2)} \tag{4.23}$$

$$= A^2(\vec{u}_1 + \alpha_2\vec{u}_2 + \alpha_3\vec{u}_3) \tag{4.24}$$

$$= \vec{u}_1 + \alpha_2\lambda_2^2\vec{u}_2 + \alpha_2\lambda_3^2\vec{u}_3 \tag{4.25}$$

Plugging in $\lambda_2 = c$ and $\lambda_3 = -c$, we see that

$$\vec{x}^{(k)} = \vec{u}_1 + c^2(\alpha_2\vec{u}_2 + \alpha_3\vec{u}_3) \tag{4.26}$$

Equations 4.22 and 4.26 allow us to solve for $\vec{u}_1$ in closed form:

$$\vec{u}_1 = \frac{\vec{x}^{(k)} - c^2\vec{x}^{(k-2)}}{1 - c^2} \tag{4.27}$$

Since our assumption that the iterate $\vec{x}^{(k-2)}$ can be expressed as a linear combination of only u_1, u_2, and u_3 does not hold in practice, Equation 4.27 gives us an approximation to the first eigenvector, to which we must continue applying Power Method iterations to reach convergence. In particular, A^2 Extrapolation eliminates error along the eigenspaces corresponding to eigenvalues of c and $-c$, leaving error along the direction of the other eigenvectors.

Results and Discussion

Figure 4.3 shows the convergence of A^2 extrapolated PageRank and the standard Power Method, where A^2 Extrapolation was applied once at iteration 4. Empirically, A^2 extrapolation speeds up the convergence of the Power Method by 18%. The initial effect of the application increases the residual, but by correctly subtracting off much of the largest non-principal eigenvectors, the convergence upon further iterations of the Power Method is sped up.

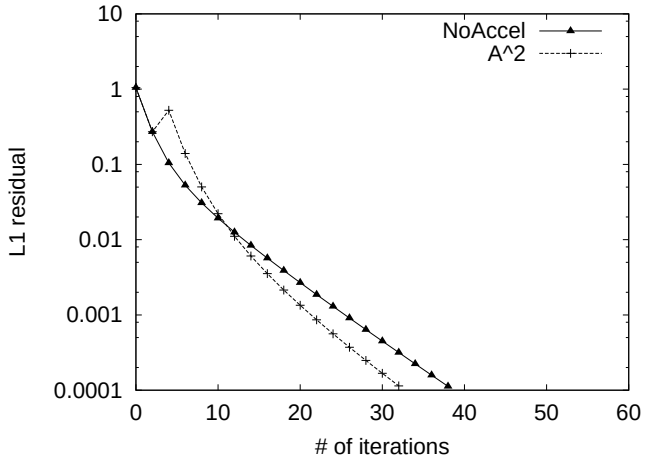


Figure 4.3: Comparison of convergence rates for Power Method and A^2 Extrapolation on LargeWeb for $c = 0.85$.

4.5.3 A^d Extrapolation

Formulation

The previous extrapolation rule made use of the fact that $(-c)^2 = c^2$. We can generalize that derivation to the case where the eigenvalues of modulus c are given by $\{c\zeta_1, \cdots, c\zeta_d\}$, where ζ_k is a dth root of unity. From Theorem 2.1 of [40] and Theorem 6 given in the Appendix, it follows that these eigenvalues arise from leaf nodes in the strongly connected component (SCC) graph of the Web that are cycles of length d. Because we know empirically that the Web has such leaf nodes in the SCC graph, it is likely that eliminating error along the dimensions of eigenvectors corresponding to these eigenvalues will speed up PageRank.

We make the assumption that $\vec{x}^{(k-d)}$ can be expressed as a linear combination of the eigenvectors $\{u_1 \ldots u_{d+1}\}$, where the eigenvalues of $\{u_2 \ldots u_{d+1}\}$ are the dth roots of unity, scaled by c.

$$\vec{x}^{(k-d)} = \vec{u}_1 + \sum_{i=2}^{d+1} \alpha_i \vec{u}_i \tag{4.28}$$

function $\vec{x}^* = PowerExtrapolation(\vec{x}^{(k-d)}, \vec{x}^{(k)})$ {
$\vec{x}^* = (\vec{x}^{(k)} - c^d\vec{x}^{(k-d)})(1 - c^d)^{-1}$;
}

Algorithm 4: Power Extrapolation

Then consider the current iterate $\vec{x}^{(k)}$; because the Power Method generates iterates by successive multiplication by A, we can write $\vec{x}^{(k)}$ as

$$\vec{x}^{(k)} = A^d\vec{x}^{(k-d)} \tag{4.29}$$

$$= A^d(\vec{u}_1 + \sum_{i=2}^{d+1} \alpha_i\vec{u}_i) \tag{4.30}$$

$$= \vec{u}_1 + \sum_{i=2}^{d+1} \alpha_i\lambda_i^d\vec{u}_i \tag{4.31}$$

$$\tag{4.32}$$

But since λ_i is cd_i, where d_i is a dth root of unity,

$$\vec{x}^{(k)} = \vec{u}_1 + c^d \sum_{i=2}^{d+1} \alpha_i\vec{u}_i \tag{4.33}$$

Equations 4.28 and 4.33 allow us to solve for $\vec{u}_1$ in closed form:

$$\vec{u}_1 = \frac{\vec{x}^{(k)} - c^d\vec{x}^{(k-d)}}{1 - c^d} \tag{4.34}$$

For instance, for $d = 4$, the assumption made is that the nonprincipal eigenvalues of modulus c are given by c, $-c$, ci, and $-ci$ (i.e., the 4th roots unity). A graph in which the leaf nodes in the SCC graph contain only cycles of length l, where l is any divisor of $d = 4$ has exactly this property.

Algorithms 4 and 5 show how to use A^d Extrapolation in conjunction with the Power Method. Note that Power Extrapolation with $d = 1$ is just Simple Extrapolation.

function $\vec{x}^{(n)} = ExtrapolatedPowerMethod(d)$ {
$\vec{x}^{(0)} = \vec{v}$;
$k = 1$;
repeat
  $\vec{x}^{(k)} = A\vec{x}^{(k-1)}$;
  $\delta = ||x^{(k)} - x^{(k-1)}||_1$;
  if $k == d + 2$,
    $\vec{x}^{(k)} = PowerExtrapolation(\vec{x}^{(k-d)}, \vec{x}^{(k)})$;
  $k = k + 1$;
until $\delta < \epsilon$;
}

Algorithm 5: Power Method with Power Extrapolation

Operation Count

The overhead in performing the extrapolation shown in Algorithm 4 comes from computing the linear combination $(\vec{x}^{(k)} - c^d\vec{x}^{(k-d)})(1 - c^d)^{-1}$, an $O(n)$ computation.

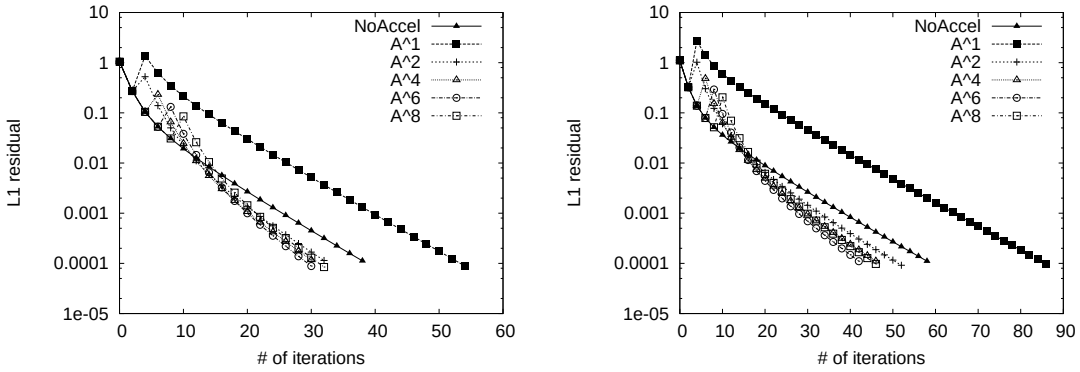
In our experimental setup, the overhead of a single application of Power Extrapolation is 1% the cost of a standard power iteration. Furthermore, Power Extrapolation needs to be applied only once to achieve the full benefit.

Results and Discussion

In our experiments, A^d Extrapolation performs the best for $d = 6$. Figure 4.4 plots the convergence of A^d Extrapolation for $d \in \{1, 2, 4, 6, 8\}$, as well as of the standard Power Method, for $c = 0.85$ and $c = 0.90$.

The wallclock speedups, compared with the standard Power Method, for these 5 values of d for $c = 0.85$ are given in Table 4.1.

For comparison, Figure 4.5 compares the convergence of the Quadratic Extrapolated PageRank with A^6 Extrapolated PageRank. Note that the speedup in convergence is similar; however, A^6 Extrapolation is much simpler to implement, and has negligible overhead, so that its wallclock-speedup is higher. In particular, each application of Quadratic Extrapolation requires 32% of the cost of an iteration, and must be applied several times to achieve maximum benefit.



(a) c=0.85 (b) c=0.90

Figure 4.4: Convergence rates for A^d Extrapolation, for $d \in \{1, 2, 4, 6, 8\}$, compared with standard Power Method.

Table 4.1: Wallclock speedups for A^d Extrapolation, for $d \in 1, 2, 4, 6, 8$, and Quadratic Extrapolation

Type	speedup
$d = 1$	-28%
$d = 2$	18%
$d = 4$	25.8%
$d = 6$	30%
$d = 8$	21.8%
Quadratic	20.8%

4.6 Related Work

Speeding up PageRank computations has become a very active field of research. Arasu et al. [3] investigate using Gauss-Seidel iterations, in which new values for pages computed early in a given iteration can be used when computing new PageRank values later in the same iteration. Chien et al. [16] and Langville and Meyer [50] suggest ways of speeding up PageRank in the context of updates, where we assume we have computed PageRank for a Web crawl, and would like to update the ranks given a newer crawl. Lee et al. [52] introduce a technique called *lumping* to speed up PageRank by treating all dangling nodes (e.g., pages with no outlinks) as a single

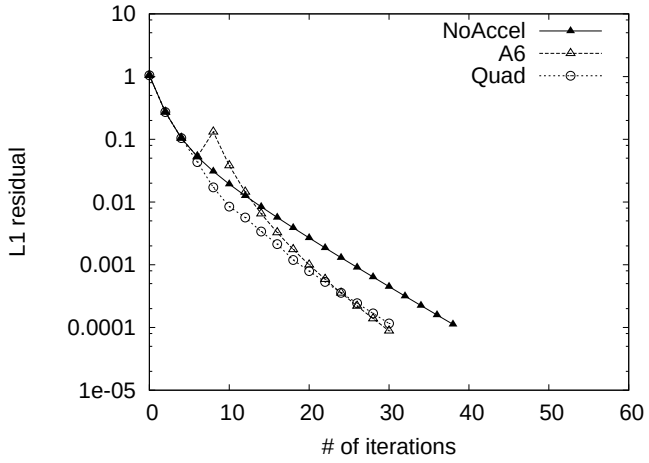


Figure 4.5: Comparison of convergence rates for Power Method, A^6 Extrapolation, and Quadratic Extrapolation on LargeWeb for $c = 0.85$.

node during part of the computation. Broder et al. suggest a technique for computing PageRank using graph aggregation, where sets of nodes are treated as supernodes; their algorithm is similar to the block-oriented algorithms we present in Chapter 5.

4.A Measures of Convergence

In this section, we present empirical results demonstrating the suitability of the L_1 residual, even in the context of measuring convergence of *induced document rankings*. In measuring the convergence of the PageRank vector, prior work has usually relied on $\delta_k = ||Ax^{(k)} - x^{(k)}||_p$, the L_p norm of the residual vector, for $p = 1$ or $p = 2$, as an indicator of convergence. Given the intended application, we might expect that a better measure of convergence is the distance, using an appropriate measure of distance, between the rank orders for query results induced by $Ax^{(k)}$ and $x^{(k)}$. We use two measures of distance for rank orders, both based on the the Kendall's-τ rank correlation measure: the KDist measure, defined below, and the $K_{\min}$ measure, introduced by Fagin et al. in [24]. To see if the residual is a "good" measure of convergence, we compared it to the KDist and $K_{\min}$ of rankings generated by $Ax^{(k)}$ and $x^{(k)}$.

We show empirically that in the case of PageRank computations, the L_1 residual δ_k is closely correlated with the KDist and $K_{\min}$ distances between query results generated using the values in $Ax^{(k)}$ and $x^{(k)}$.

Our distance measure KDist is defined in terms of the KSim similarity measure we introduced in Section 3.3.1. Consider two partially ordered lists of URLs, τ_1 and τ_2, each of length k. We define KDist as

$$\text{KDist}(\tau_1, \tau_2) = 1 - \text{KSim}(\tau_1, \tau_2) \tag{4.35}$$

In other words, $\text{KDist}(\tau_1, \tau_2)$ is the probability that the the two ordered lists *disagree* on the relative order of a pair of URLs.

To measure the convergence of PageRank iterations in terms of induced rank orders, we measured the KDist distance between the induced rankings for the top 100 results, averaged across 27 test queries, using successive power iterates for the LargeWeb dataset, with the damping factor c set to 0.9.[8] The average residuals

[8]Computing Kendall's τ over the complete ordering of all of LargeWeb is expensive; instead we opt to compute KDist and $K_{\min}$ over query results.

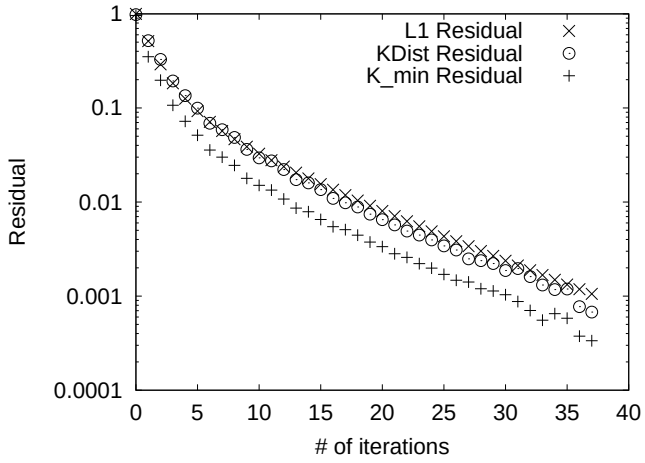


Figure 4.6: Comparison of the L_1 residual, KDist and K_{min} for PageRank iterates. Note that the two curves nearly perfectly overlap, suggesting that in the case of PageRank, the easily calculated L_1 residual is a good measure for the convergence of query-result rankings.

using the KDist, K_{min}, and L_1 measures are plotted in Figure 4.6.[9] Surprisingly, the L_1 residual is almost perfectly correlated with KDist, and is closely correlated with K_{min}.[10] A rigorous explanation for the close match between the L_1 residual and the Kendall's τ based residuals is an interesting avenue of future investigation.

[9]The L_1 residual δ_k is normalized so that δ_0 is 1.

[10]We emphasize that we have shown close agreement between L_1 and KDist for measuring residuals, not for distances between arbitrary vectors.

4.B External Theorems

For convenience, we include in this appendix theorems that are proven elsewhere that are cited in this chapter.

Theorem 3 *(from page 126 of [44]) If P is the transition matrix for a finite Markov chain, then the multiplicity of the eigenvalue 1 is equal to the number of irreducible closed subsets of the chain.*

Theorem 4 *(from page 4 of [73]) If $\vec{x_i}$ is an eigenvector of A corresponding to the eigenvalue λ_i, and $\vec{y_j}$ is an eigenvector of A^T corresponding to λ_j, then $\vec{x_i}^T \vec{y_j} = 0$ (if $\lambda_i \neq \lambda_j$).*

Theorem 5 *(from page 82 of [43]) Two distinct states belonging to the same class (irreducible closed subset) have the same period. In other words, the property of having period d is a class property.*

Theorem 6 *(Theorem IV.2.5 from [44]) If P is the transition matrix of an irreducible periodic Markov chain with period d, then the dth roots of unity are eigenvalues of P. Further, each of these eigenvalues is of multiplicity one and there are no other eigenvalues of modulus 1.*

Chapter 5

Block-Oriented PageRank Computation

5.1 Introduction[1]

In this chapter, we consider block-oriented approaches to speeding up the computation of PageRank. We present two block-oriented algorithms; the first makes no assumptions about the empirical properties of the Web's graph structure, but uses a block-oriented technique for memory-efficient computation; the second, BlockRank, exploits the inherent "block structure" of the Web link graph to further speed up the computation. Block structure refers to the tendency of pages to link to other pages on the same host. Analysis of the graph structure of the Web has concentrated on determining various properties of the graph, such as degree distributions and connectivity statistics, and on modeling the creation of the Web graph [5]. However, this research has not directly addressed how this inherent structure can be exploited effectively to speed up link analysis. Raghavan and Garcia-Molina [67] have exploited the hostname (or more generally, URL)-induced structure of the Web to represent the Web graph efficiently. BlockRank directly exploits this kind of structure to achieve speedups for computing PageRank by

[1] This chapter covers joint work we first presented in [33, 46]

- Substantially improving locality of reference, thereby reducing disk i/o costs and memory access costs,
- Reducing the computational complexity (i.e., number of FLOPS).
- Allowing for highly parallelizable computations requiring little communication overhead,
- Allowing reuse of previous computations when updating PageRank and when computing multiple topic-sensitive PageRank vectors.

5.2 The Naive Algorithm for Computing PageRank

The implementation of PageRank on small graphs is simple. Computing PageRank on Web-scale link-graphs, however, requires much greater care in the use of data structures. We begin with a detailed discussion of a naive implementation to gain a clear understanding of the system issues involved for matrix-vector multiplications in the specific context of PageRank.

The link structure for the Web graph, referred to as $\mathcal{L}$, is stored on disk in a binary format, illustrated textually in Figure 5.1.

Source Node (32-bit id)	Out Degree (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	**4**	**12, 26, 58, 94**
1	**3**	**15, 56, 81**
2	**5**	**9, 10, 38, 45, 78**

Figure 5.1: Depiction of the datastructure holding the Web hyperlink graph.

The source-id and each of the destination-id's are stored as 32-bit integers. The outdegree is stored as a 16-bit integer. For an 81-million node dataset (described in

$$\begin{pmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{pmatrix} = c \begin{pmatrix} L_{11} & . & . & . & L_{1n} \\ . & . & . & . & . \\ L_{j1} & . & L_{jk} & . & L_{jn} \\ . & . & . & . & . \\ L_{n1} & . & . & . & L_{nn} \end{pmatrix} \times \begin{pmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{pmatrix} + (1-c) \begin{pmatrix} v_1 \\ . \\ . \\ . \\ v_n \end{pmatrix}$$

Figure 5.2: Matrix-vector multiplication corresponding to a PageRank iteration. This figure illustrates Algorithm 1 that was given in Section 3.1.1. If page k links to page j, then L_{jk} is $1/\deg(k)$. Note that L is very sparse, since most pages have only a few outlinks. For unpersonalized computations, $\vec{v}$ is uniform, in which case we do not materialize it explicitly so that it requires no memory. For personalized or topic-sensitive computations, $\vec{v}$ generally has on the order of thousands of nonzero entries, in which case it is sparse enough to require negligible memory. $\vec{x}$ and $\vec{y}$ are dense, and must be maintained explicitly, either in memory or on disk.

Section 5.3.1) the size of the link structure is 1.01 GB, and is assumed to exceed the size of main memory. Although modern machines can contain several gigabytes of memory, the Web has grown to billions of pages, so that this assumption is still valid.

The setup for the naive PageRank implementation is as follows. We create two arrays of floating point values representing the source and destination rank vectors, denoted by $\vec{x}$ and $\vec{y}$ resp., as shown in the matrix-vector multiplication given in Figure 5.2. Each vector has n entries, where n is the number of nodes in our Web graph. The naive implementation of the PageRank computation[2] is given as Algorithm 6.

We make successive passes over $\mathcal{L}$, using the current rank values held in $\vec{x}$, to compute the rank values for the subsequent iteration, held in $\vec{y}$. We can stop iterating when the *residual*, defined as the norm of the difference between $\vec{x}$ and $\vec{y}$, reaches some threshold. Recall from Appendix 4.A that the residual is a good proxy for measuring the convergence of the document rankings induced by the successive PageRank iterates.

Assuming main memory is large enough to hold $\vec{x}$ and $\vec{y}$, the i/o cost for each

[2]See Section 3.1.1 of Chapter 3.

```
function NaivePageRank(L, v, c) {
x = v;
while δ > τ do
    y = 0_n;
    while not L.eof() do
        L.read(source, m, dest_1, dest_2, ..., dest_m);
        for j = 1 ... m do
            y_{dest_j} = y_{dest_j} + (c/m) x_{source};
        end
    end
    // use a sparse representation for v
    y = y + (1 − c)v ;
    // compute the following only periodically
    δ = ||x − y||_1 ;
    x = y;
end
}
```

Algorithm 6: Naive PageRank algorithm. As first introduced in Section 3.1.1, $\vec{v}$ is the personalization vector, and c is the damping factor.

iteration of the above implementation is given by:

$$C = \textbf{sizeof}(\mathcal{L})$$

If main memory is large enough to hold only $\vec{y}$, and we assume that the link structure is sorted on the source field, the i/o cost is given by:

$$C = \textbf{sizeof}(\vec{x}) + \textbf{sizeof}(\vec{y}) + \textbf{sizeof}(\mathcal{L})$$

$\vec{x}$ needs to be read sequentially from disk during the rank propagation step, and $\vec{y}$ needs to be written to disk to serve as the source vector for the subsequent iteration.

A large Web crawl with billions of pages will result in rank vectors that exceed the main memory of most computers. As mentioned above, if the link structure is sorted on the source field, the accesses on $\vec{x}$ will be sequential, and will not pose a problem. However, the random access pattern on $\vec{y}$ leads the working set of this

$$\begin{pmatrix} y_1 \\ . \\ y_{l-1} \end{pmatrix} = c \begin{pmatrix} L_{11} & . & . & . & L_{1n} \\ . & . & . & . & . \\ L_{(l-1),1} & . & . & . & L_{(l-1),n} \end{pmatrix} \times \begin{pmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{pmatrix} + (1-c) \begin{pmatrix} v_1 \\ . \\ v_{l-1} \end{pmatrix}$$

$$\begin{pmatrix} y_l \\ . \\ y_{m-1} \end{pmatrix} = c \begin{pmatrix} L_{l1} & . & . & . & L_{ln} \\ . & . & . & . & . \\ L_{(m-1),1} & . & . & . & L_{(m-1),n} \end{pmatrix} \times \begin{pmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{pmatrix} + (1-c) \begin{pmatrix} v_l \\ . \\ v_{m-1} \end{pmatrix}$$

$$\begin{pmatrix} y_m \\ . \\ y_n \end{pmatrix} = c \begin{pmatrix} L_{m1} & . & . & . & L_{mn} \\ . & . & . & . & . \\ L_{n1} & . & . & . & L_{nn} \end{pmatrix} \times \begin{pmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{pmatrix} + (1-c) \begin{pmatrix} v_m \\ . \\ v_n \end{pmatrix}$$

Figure 5.3: Block-oriented multiplication with 3 blocks. Each block-multiply requires a full pass over $\vec{x}$, but only involves 1/3 of $\vec{y}$, L, and $\vec{v}$.

implementation to equal **sizeof**($\vec{y}$). As we will see in Section 5.3.1, if main memory cannot accommodate $\vec{y}$, the running time will increase dramatically and the above cost analysis becomes invalid.

5.3 Memory-Efficient Computation of PageRank

We now describe how we can control the working set of the PageRank algorithm by partitioning the destination vector $\vec{y}$, the cause of the large working set, into β blocks each of size b, as illustrated in Figure 5.3.[3] The links file $\mathcal{L}$ must be rearranged to reflect this setup. We partition $\mathcal{L}$ into β links files $\mathcal{L}_0, \ldots, \mathcal{L}_{\beta-1}$, such that the *dest* field in $\mathcal{L}_i$ contains only nodes in the set $\{u | b \times i \leq u < b \times (i+1)\}$. In other words, the outgoing links of a node are bucketed according to the range that the identifier

[3]If the number of pages n is not divisible by β, the final block of $\vec{y}$ can be padded.

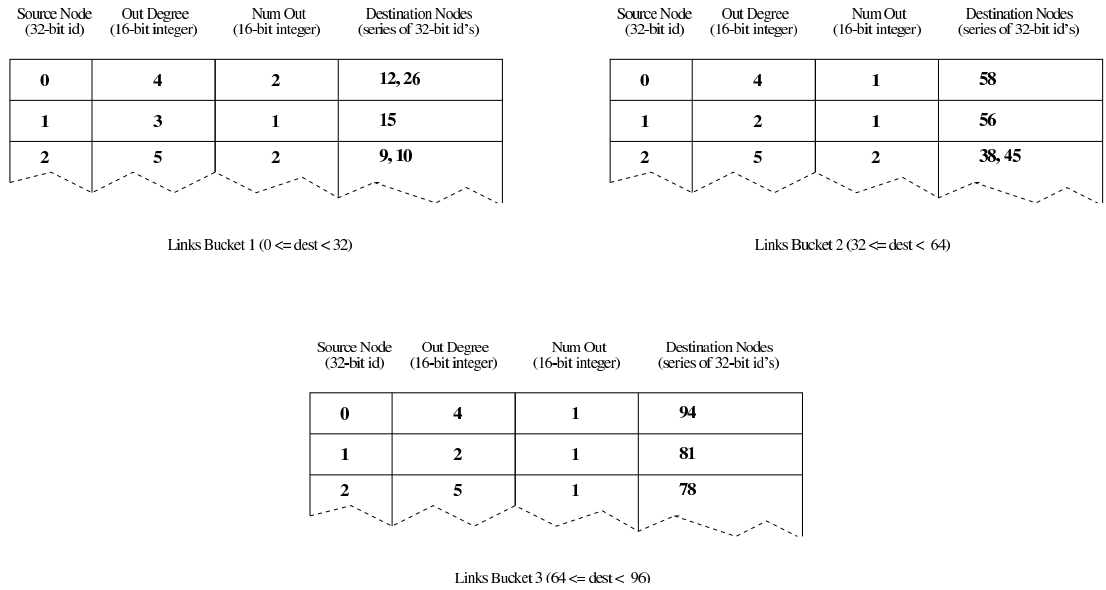


Figure 5.4: Partitioned link file.

of the destination page falls into. The partitioning of the links of Figure 5.1 for the case of $\beta = 3$ is shown in Figure 5.4. Note that $\sum \mathbf{sizeof}(\mathcal{L}_i) > \mathbf{sizeof}(\mathcal{L})$ because of the extra overhead caused by the redundant storage of the source node and outdegree entries in each of the partitions. The block-oriented algorithm is given as Algorithm 7.

Because $\mathcal{L}_i$ is sorted on the *source* field, each pass through $\mathcal{L}_i$ requires only one sequential pass through $\vec{x}$. By choosing a large enough β, we can ensure the working set of the algorithm fits in main memory, so that no swapping occurs. Define ϵ to be such that

$$\sum_i \mathbf{sizeof}(\mathcal{L}_i) = (1+\epsilon) \cdot \mathbf{sizeof}(\mathcal{L})$$

The i/o cost of this approach is then given by:

$$C = \beta \cdot \mathbf{sizeof}(\vec{x}) + \mathbf{sizeof}(\vec{y}) + (1+\epsilon) \cdot \mathbf{sizeof}(\mathcal{L})$$

In practice, ϵ is reasonably small, as shown in Table 5.2. The other cost introduced by the partitioning scheme is the need to make β passes over the source vector $\vec{x}$.

```
function MemEffPageRank(L, v, c) {
x = v ;
while δ > τ do
    for i = 0 ... β − 1 do
        y^(i) = 0_b ;
        while not L_i.eof() do
            L_i.read(source, m, k, dest_1, dest_2, ..., dest_k) ;
            for j = 1 ... k do
                y^(i)_dest_j = y^(i)_dest_j + (c/m) x_source ;
            end
        end
        // use a sparse representation for v
        y^(i) = y^(i) + (1 − c) v^(i) ;
        OutputToDisk(y^(i)) ;
    end
    // compute the following only periodically
    δ = ||x − y||_1 ;
    x = y ;
end
}
```

Algorithm 7: Memory-efficient PageRank algorithm.

However, since in practice we have $\textbf{sizeof}(\mathcal{L}) > 5 \cdot \textbf{sizeof}(\vec{x})$, this additional overhead is reasonable. Note that computing the residual during every iteration would require an additional pass over $\vec{x}$, which is not included in the above cost analysis. We can largely avoid this cost by computing the residual only at fixed intervals.

If we had stored the links in transpose format $\mathcal{L}^T$, in which each entry contained a node and a list of parent nodes, then the above algorithm would remain essentially the same, except that we would break the source vector $\vec{x}$ into β blocks, and make multiple passes over the destination vector $\vec{y}$. We would successively load in blocks of $\vec{x}$, and fully distribute its rank according to $\mathcal{L}^T$ to all destinations in $\vec{y}$. However, note that each "pass" over $\vec{y}$ requires reading in the values from disk, adding in the current source block's contributions, and then writing out the updated values to disk. Thus, using $\mathcal{L}^T$ rather than $\mathcal{L}$ incurs an additional i/o cost of $\textbf{sizeof}(\vec{y})$, since $\vec{y}$ is both read and written on each pass.

5.3.1 Experimental Results

At the time of these experiments, the Stanford WebBase, our local repository of the Web, contained roughly 25 million pages. There are roughly 81 million URLs in the corresponding link graph, including URLs that were not themselves crawled, but exist in the bodies of crawled pages. For our experiments, we first used a preprocessing step that removed *dangling* pages, meaning pages with no outlinks. Starting with the 81-million-node graph, all nodes with outdegree 0 were removed. The step was repeated once more on the resulting graph, yielding a subgraph with close to 19 million nodes. This process was used since the original graph is a truncated snapshot of the Web with many dangling nodes. The node id's were assigned consecutively from 0 to 18,922,290.

We used a 450MHz Pentium-III machine with a 7200-RPM Western Digital Caviar AC418000 hard disk. We measured the running times of PageRank over roughly 19 million pages using three different partitionings: 1-block (i.e., naive), 2-blocks, and 4-blocks. The expected memory usage is given in Table 5.1. We tested the three partitioning strategies on three different memory configurations: 256 MB, 64 MB, and 32 MB.

Number of Blocks	Expected Working Set
1	72 MB
2	36 MB
4	18 MB

Table 5.1: Expected memory usage for different numbers of blocks.

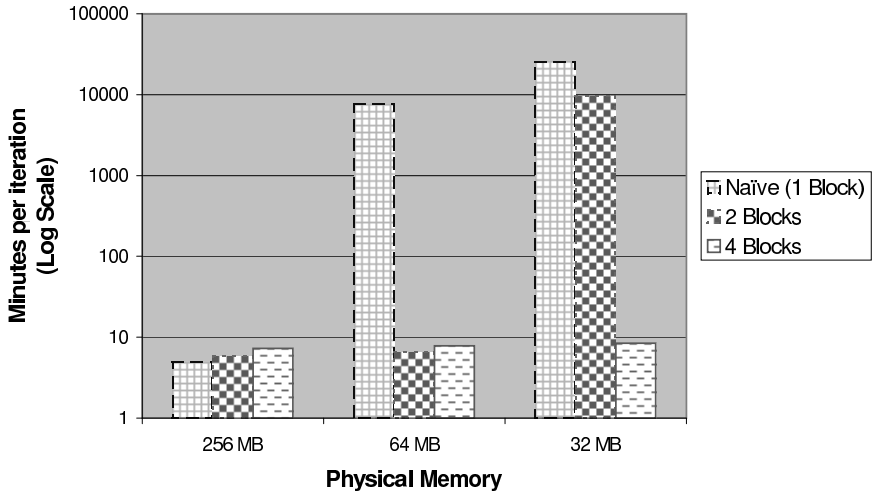


Figure 5.5: Log plot of running times.

The time required per iteration of PageRank is given for each of the three partitionings under each of the three memory configurations in Figure 5.5. As expected, the most efficient strategy is to partition the destination vector $\vec{y}$ into enough blocks so that a single block of $\vec{y}$ can fit in physical memory. Using too many blocks slightly degrades performance, as both the number of passes over $\vec{x}$ and the size of $\mathcal{L}$ increase. Table 5.2 shows the total size of the link structure for the three partitionings, as well as the associated ϵ, as discussed in Section 5.3. Using too few blocks, however, degrades performance by several orders of magnitude. For the cases in Figure 5.5 where the block size exceeds physical memory, we had to estimate the full iteration time from a partially completed iteration.

The block-oriented strategy is very effective in controlling the memory requirements of PageRank computations. This block-oriented PageRank is not an approximation of normal PageRank: the same matrix-vector multiplication is performed

Number of Blocks	Size of $\mathcal{L}$	ϵ
1	1.01 GB	0
2	1.14 GB	0.13
4	1.29 GB	0.28

Table 5.2: Overhead of partitioned link structure.

whether or not $\mathcal{L}$ is partitioned. For billions of pages, the block-oriented technique is essential in computing PageRank, even on machines with fairly large amounts of main memory.

5.4 Exploiting the Inherent Block Structure of the Web

In the previous section, we did not exploit any of the inherent properties of the Web graph. In this section, we take a detailed look at the structure of the Web graph, and introduce an algorithm, BlockRank, for efficiently computing PageRank by exploiting this structure.

5.4.1 Description of Datasets

We begin with a description of the two datasets we used for this set of experiments. The STANFORD/BERKELEY link graph was generated from a crawl of the stanford.edu and berkeley.edu domains done in December 2002 by the Stanford WebBase project. This link graph (after dangling node removal, discussed below) contains roughly 683,500 nodes, with 7.6 million links, and requires 25MB of storage. We used STANFORD/BERKELEY while developing the algorithms, to get a sense for their expected performance. For real-world performance measurements, we use the LARGEWEB link graph, generated from a crawl of the Web done by the Stanford WebBase project in January 2001 [41]. The full version of this graph, termed FULL-LARGEWEB, contains roughly 290M nodes, just over a billion edges, and requires 6GB of storage. Many of these nodes are dangling nodes (pages with no outlinks), either because the pages genuinely have no outlinks, or because they are pages that have been discovered

Term	Example: cs.stanford.edu/research/
top level domain	edu
domain	stanford.edu
hostname	cs
host	cs.stanford.edu
path	/research/

Table 5.3: Example illustrating our terminology using the sample URL http://cs.stanford.edu/research/.

		Domain		Host	
Full	*Intra*	953M links	83.9%	899M links	79.1%
	Inter	183M links	16.1%	237M links	20.9%
DNR	*Intra*	578M links	95.2%	568M links	93.6%
	Inter	29M links	4.8%	39M links	6.4%

Table 5.4: Hyperlink statistics on LargeWeb for the full graph (*Full*: 291M nodes, 1.137B links) and for the graph with dangling nodes removed (*DNR*: 64.7M nodes, 607M links).

but not crawled. We also consider the version of LargeWeb with dangling nodes removed, termed DNR-LargeWeb, which contains roughly 70M nodes, with over 600M edges, and requires 3.6GB of storage.

5.4.2 Block Structure of the Web

The key terminology we use in the remaining discussion is given in Table 5.3. To investigate the structure of the Web, we performed the following simple experiment. We took all the hyperlinks in Full-LargeWeb, and counted how many of these links are "intra-host" links (links from a page to another page in the same host) and how many are "inter-host" links (links from a page to a page in a different host). Table 5.4 shows that 79.1% of the links in this dataset are intra-host links, and 20.9% are inter-host links. These intra-host connectivity statistics are not far from the earlier results of Bharat et al. [5]. We also investigated the number of links that are intra-domain links, and the number of links that are inter-domain links. Notice in Table 5.4 that an even larger majority of links are intra-domain links (83.9%).

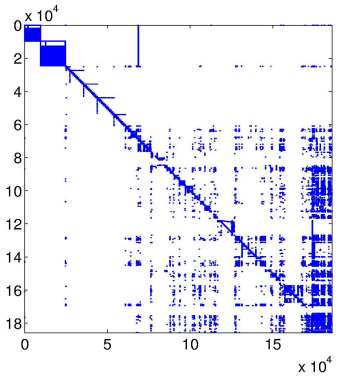
These results are intuitive. Take as an example the domain cs.stanford.edu. Most of the links in cs.stanford.edu are near the cs.stanford.edu site (such as cs.stanford.edu /admissions, or cs.stanford.edu/research). Furthermore, almost all non-navigational links are links to other Stanford hosts, such as www.stanford.edu, library.stanford.edu, or www-cs-students.stanford.edu.

One might expect this structure exists in lower levels of the Web hierarchy as well. For example, one might expect that pages under cs.stanford.edu/admissions/ are highly interconnected, and very loosely connected with pages in other sublevels, leading to a nested block structure. This type of nested block structure can be naturally exposed by sorting the link graph to construct a link matrix in the following way. We sort URLs lexicographically, except that as the sort key, we reverse the components of the domain. For instance, the sort key for the URL www.stanford.edu/home/students/ would be edu.stanford.www/home/students. The URLs are then assigned sequential identifiers when constructing the link matrix. A link matrix contains as its (i, j)th entry a 1 if there is a link from i to j, and 0 otherwise. This has the desired property that URLs are grouped in turn by top level domain, domain, hostname, and finally path. The subgraph for pages in stanford.edu appear as a sub-block of the full link matrix. In turn, the subgraph for pages in www-db.stanford.edu appear as a nested sub-block.

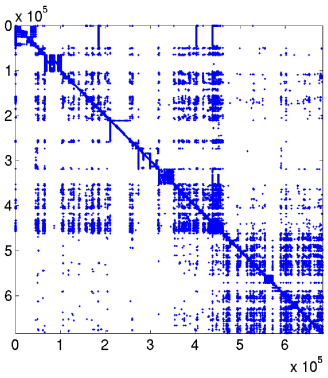
We can then gain insight into the structure of the Web by using dotplots to visualize the link matrix. In a dotplot, if there exists a link from page i to page j then point (i, j) is colored; otherwise, point (i, j) is white. Since our full datasets are too large to see individual pixels, we show several slices of the Web in Figure 5.6. Notice three things:

1. There is a definite block structure to the Web.
2. The individual blocks are much smaller than entire Web.
3. There are clear nested blocks corresponding to domains, hosts, and subdirectories within the path.

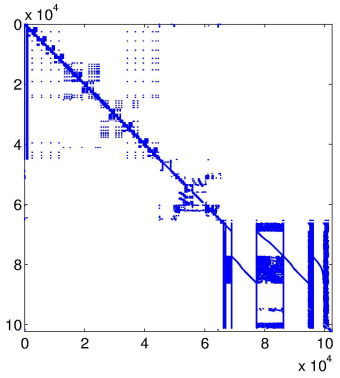
Figure 5.6(a) shows the dotplot for the ibm.com domain. Notice that there are clear blocks, which correspond to different hosts within ibm.com; for example, the



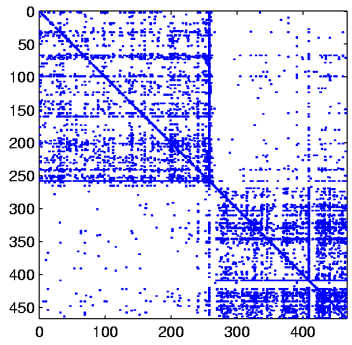
(a) IBM



(b) Stanford/Berkeley



(c) Stanford-50



(d) Stanford/Berkeley Host Graph

Figure 5.6: A view of 4 different slices of the Web: (a) the IBM domain, (b) all of the hosts in the Stanford and Berkeley domains, (c) the first 50 Stanford hosts, alphabetically, and (d) the host-graph of the Stanford and Berkeley domains.

upper left block corresponds to the almaden.ibm.com hosts (the hosts for IBM's Almaden Research Center). Notice that the pages at the very end of the plot (pages $i \geq 18544$) are heavily inlinked (the vertical line at the lower right hand corner of the plot. These are the pages within the www.ibm.com host, and it is expected that they be heavily inlinked.

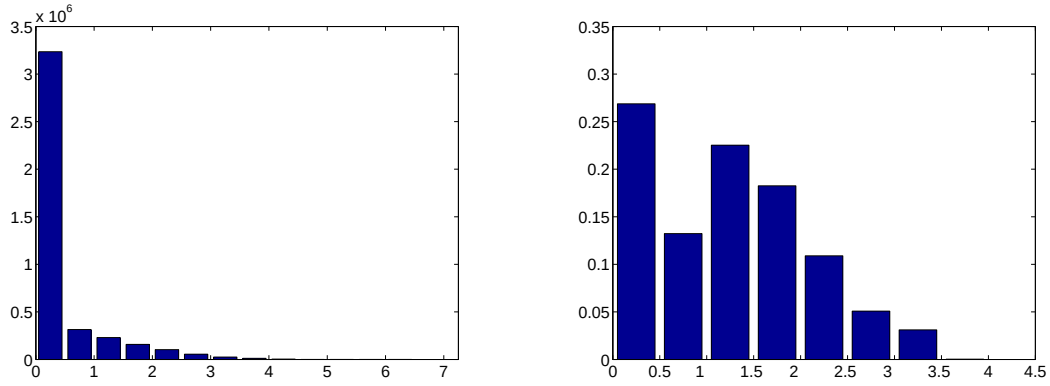
Figure 5.6(b) shows the dotplot for STANFORD/BERKELEY. Notice that this also has a strong block structure and a dense diagonal. Furthermore, this plot makes clear the nested block structure of the Web. The superblock on the upper left hand side is the stanford.edu domain, and the superblock on the lower right hand side is the berkeley.edu domain.

Figure 5.6(c) shows a closeup of the first 50 hosts alphabetically within the stanford.edu domain. The majority of this dotplot is composed of 3 hosts that are large: acomp.stanford.edu, the academic computing site at Stanford, in the upper left hand corner; cmgm.stanford.edu, an online bioinformatics resource, in the middle, and daily.stanford.edu, the Web site for the *Stanford Daily* (Stanford's student newspaper) in the lower right hand corner.

Figure 5.6(d) shows the host graph for the stanford.edu and berkeley.edu domains, in which each host is treated as a single node, and an edge is placed between host i and host j if there is a link between any page in host i and host j. Again, we see strong block structure on the domain level, and the dense diagonal shows strong block structure on the host level as well. The vertical and horizontal lines near the bottom right hand edge of both the Stanford and Berkeley domains represent the www.stanford.edu and www.berkeley.edu hosts, showing that these hosts are, as expected, strongly linked to most other hosts within their own domain.

Block Sizes

We investigate next the sizes of the hosts in the Web. Figure 5.7(a) shows the distribution over number of (crawled) pages of the hosts in LARGEWEB. Notice that the majority of sites are small, on the order of 10^0 pages. Figure 5.7(b) shows the sizes of the host blocks in the Web when dangling nodes are removed. When dangling nodes are removed, the blocks become smaller, and the distribution is still skewed



(a) FULL-LARGEWEB (b) DNR-LARGEWEB

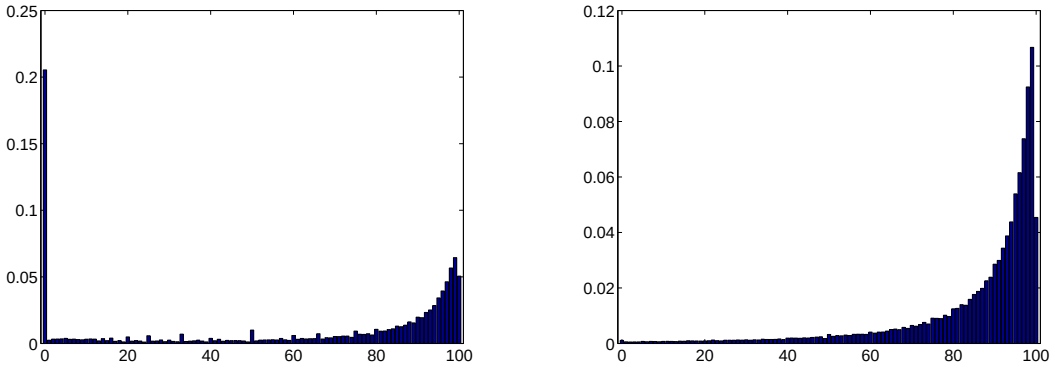
Figure 5.7: Histogram of distribution over host sizes of the Web. The x-axis gives bucket sizes for the $\log_{10}$ of the size of the host-blocks, and the y-axis gives the fraction of host-blocks that are that size.

towards small blocks. The largest block had 6,000 pages. In future sections we see how to exploit the small sizes of the blocks, relative to the dataset as a whole, to speedup up link analysis.

The GeoCities Effect

Intra-host link density refers to the fraction of outlinks in a host are to other pages on the same host. While one would expect that most hosts have high intra-host link density, as in cs.stanford.edu, there are some hosts that one would expect to have low intra-host link density, for example pages.yahoo.com (formerly www.geocities.com). The Web site http://pages.yahoo.com is the root page for Yahoo! GeoCities, a free Web hosting service. There is no reason to think that people who have Web sites on GeoCities would prefer to link to one another rather than to sites not in GeoCities.[4] Figure 5.8 shows two histograms of the intra-host densities of the Web. These histograms shows how many hosts have a given intra-host density. In Figure 5.8(a) there is a spike near 0% intra-host linkage, showing that there are many hosts that do not have high intra-host linkage, relative to their *inter*-host linkage (e.g., the Geocities

[4]There may of course be deeper structure found in the path component, although we currently do not directly exploit such structure.



(a) All hosts (b) Hosts with < 5 pages excluded

Figure 5.8: Distribution over intra-host outlink density of host blocks for the DNR-LargeWeb data set. The x-axis of each figure shows percentile buckets for intra-host linkage density (the percent of edges originating or terminating in a given host that are intra-host links), and the y-axis shows the fraction of hosts that have that linkage density. Figure 5.8(a) shows the distribution of intra-host linkage density for all hosts; 5.8(b) shows it for all hosts that have 5 or more pages.

Effect). However, when we remove the hosts that have only 1 page (Figure 5.8(b)), this spike is substantially dampened, and when we exclude hosts with fewer than 5 pages, the spike is eliminated. This shows that the hosts in LargeWeb that are not highly intraconnected are very small in size. When the very small hosts are removed, the great majority of remaining hosts have high intra-host densities, and very few of them suffer from the GeoCities effect.

5.4.3 BlockRank Algorithm

We now present the BlockRank algorithm that exploits the empirical findings of the previous section to speed up the computation of PageRank. This work is motivated by and builds on aggregation/disaggregation techniques [19, 71] and domain decomposition techniques [25] in numerical linear algebra. Steps 2 and 3 of the BlockRank algorithm are similar to the Rayleigh-Ritz refinement technique [54].

Overview of BlockRank Algorithm

The block structure of the Web suggests a fast algorithm for computing PageRank, wherein a "local PageRank vector" is computed for the pages on each host *independently*, giving the relative importance of pages within a host. These local PageRank vectors can then be used to form an approximation to the global PageRank vector that is used as a starting vector for the standard PageRank computation (e.g., the starting vector for the Power Method). This is the basic idea behind the BlockRank algorithm, which we summarize here and describe in this section. The algorithm proceeds as follows:

0. Split the Web into blocks by domain.
1. Compute the Local PageRanks for each block (Section 5.4.3).
2. Estimate the relative importance, or "BlockRank" of each block (Section 5.4.3).
3. Weight the Local PageRanks in each block by the BlockRank of that block, and concatenate the weighted Local PageRanks to form an approximate Global PageRank vector $\vec{z}$ (Section 5.4.3).
4. Use $\vec{z}$ as a starting vector for standard PageRank (Section 5.4.3).

We describe the steps in detail below, and we introduce some notation here. We will use lower-case indices (i.e. i, j) to represent indices of individual Web sites, and upper case indices (i.e. I, J) to represent indices of blocks. We use the shorthand notation $i \in I$ to denote that page i is in the set of pages corresponding to block I. The number of elements in block J is denoted n_J. The graph of a given block J is given by the $n_J \times n_J$ submatrix G_{JJ} of the matrix G.

Computing Local PageRanks

In this section, we describe computing a "local PageRank vector" for each block in the Web. Since most blocks have very few links in and out of the block as compared to the number of links within the block, it seems plausible that the relative rankings of most of the pages within a block are determined by the intra-block links.

We define the *local PageRank vector* $\vec{l}_J$ of a block J (G_{JJ}) to be the result of the PageRank algorithm applied only on block J, as if block J represented the entire Web, and as if the links to pages in other blocks did not exist. That is:

$$\vec{l}_J = pageRank(G_{JJ}, \vec{s}_J, \vec{v}_J)$$

where the start vector $\vec{s}_J$ is the $n_J \times 1$ uniform probability vector over pages in block J ($[\frac{1}{n_J}]_{n\times 1}$), and the personalization vector $\vec{v}_J$ is the $n_J \times 1$ vector whose elements are all zero except the element corresponding to the root node of block J, whose value is 1.

In Figure 5.9, we show a histogram of the number of iterations it takes for the local PageRank scores for each host in DNR-LargeWeb to converge to an L_1 residual $< 10^{-1}$. Notice that most hosts converge to this residual in less than 12 iterations. Interestingly, there is no correlation between the convergence rate of a host and the host's size. Rather, the convergence rate is primarily dependent on the extent of the nested block structure within the host. That is, hosts with strong nested blocks are likely to converge slowly. Hosts with a more random connection pattern converge faster.

This observation suggests that one could make the local PageRank computations even faster by wisely choosing the blocks. That is, if a host has a strong nested block structure, use the directories within that host as your blocks. However, we did not pursue the idea, since the cost of the local PageRank computations is not a bottleneck for computing PageRank with our scheme, as we will discuss in Section 5.4.6.

Estimating the Relative Importance of Each Block

In this section, we describe computing the relative importance, or *BlockRank*, of each block. Assume there are k blocks in the Web. To compute BlockRanks, we first create the *block graph* B, where each vertex in the graph corresponds to a block in the Web graph. An edge between two pages in the Web is represented as an edge between the corresponding blocks (or a self-edge, if both pages are in the same block). The edge weights are determined as follows: the weight of an edge between blocks I and J is

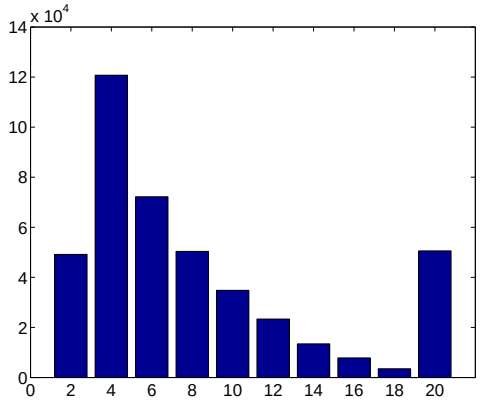


Figure 5.9: Local PageRank convergence rates for hosts in DNR-LargeWeb. The x-axis is the number of iterations until convergence to a tolerance of 10^{-1}, and the y-axis is the fraction of hosts that converge in a given number of iterations.

defined to be the sum of the edge-weights from pages in I to pages in J in the Web graph, weighted by the local PageRanks of the linking pages in block I.

That is, if A is the Web graph and l_i is the local PageRank of page i in block I, then weight of edge B_{IJ} is given by:

$$B_{IJ} = \sum_{i \in I, j \in J} A_{ij} \cdot l_i$$

We can write the above equation in matrix notation as follows. Define the local PageRank matrix L to be the $n \times k$ matrix whose columns are the local PageRank vectors $\vec{l_J}$.

$$L = \begin{pmatrix} \vec{l_1} & \vec{0} & \cdots & \vec{0} \\ \vec{0} & \vec{l_2} & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{0} & \vec{0} & \cdots & \vec{l_K} \end{pmatrix}$$

Define the matrix S to be the $n \times k$ matrix that has the same structure as L, but whose nonzero entries are all replaced by 1. Then the block matrix B is the $k \times k$

matrix:

$$B = L^T AS$$

Notice that B is a transition matrix where the element B_{IJ} represents the transition probability of block I to block J. That is:

$$B_{IJ} = p(J|I)$$

Once we have the $k \times k$ transition matrix B, we may use the standard PageRank algorithm on this reduced matrix to compute the BlockRank vector $\vec{b}$. That is:

$$\vec{b} = pageRank(B, \vec{v}_k, \vec{v}_k)$$

where $\vec{v}_k$ is the uniform k-vector $[\frac{1}{k}]_{k\times 1}$. This PageRank computation will yield the stationary distribution of the transition matrix $c \cdot B + (1-c)E_k$, where we define $E_k = [1]_{k\times 1} \times \vec{v_k}^T$. In terms of the random surfer model, we imagine a random surfer going from block to block according to the transition probability matrix B. At each stage, the surfer will get bored with probability $1-c$ and jump to a different *block*.

Approximating Global PageRank using Local PageRank and BlockRank

In this section, we find an estimate to the global PageRank vector $\vec{x}$. At this point, we have two sets of rankings. Within each block J, we have the local PageRanks $\vec{l}_J$ of the pages in the block. We also have the BlockRank vector $\vec{b}$ whose elements b_J are the BlockRank for each block J, measuring the relative importance of the blocks. We approximate the global PageRank of a page $j \in J$ as its local PageRank l_j, weighted by the BlockRank b_J of the block in which it resides. That is,

$$x_j^{(0)} = l_j \cdot b_J$$

In matrix notation, this is:

$$\vec{x}^{(0)} = L\vec{b}$$

Recall that the local PageRanks of each block sum to 1. Also, the BlockRanks

sum to 1. Therefore, our approximate global PageRanks will also sum to 1. We use our approximate global PageRank vector $\vec{x}^{(0)}$ as a start vector for the standard PageRank algorithm.

Using This Estimate as a Start Vector

In order to compute the true global PageRank vector $\vec{x}$ from our approximate PageRank vector $\vec{x}^{(0)}$, we simply use it as a start vector for standard PageRank. That is:

$$\vec{x} = pageRank(G, \vec{x}^{(0)}, \vec{v})$$

where G is the graph of the Web, and $\vec{v}$ is the uniform distribution over root nodes.

In Section 5.4.4, we show how to compute different personalizations quickly once $\vec{x}$ has been computed.

The BlockRank algorithm for computing PageRank, presented in the preceding sections, is summarized by Algorithm 8.

5.4.4 Personalized BlockRank

In this section, we describe how we can use the BlockRank algorithm and a restriction on the jump behavior of the random surfer to further reduce the computation time when computing a large number of personalized or topic-sensitive PageRank vectors. Although the standard BlockRank algorithm can be used to speed up the computation of each of the personalized PageRank vectors in isolation, we can exploit the redundant computations *across* different personalizations to speed up the process further.

As we described earlier in Section 3.1.1, personalizing the PageRank computation normally involves choosing a distribution $\vec{v}$ over pages that specifies where the random surfer jumps to when he decides to teleport rather than following an outlink. We can make the personalized computations more efficient if we instead specify a personalization vector over *hosts* that the random surfer jumps to. For example, when teleporting, a random surfer interested in sports may jump to the www.espn.com host, but he may not jump to http://www.espn.com/ncb/columns/forde_pat/index.html. We

0. Sort the Web graph lexicographically as described in Section 5.4.2, exposing the nested block structure of the Web.
1. Compute the local PageRank vector $\vec{l}_J$ for each block J.

foreach *block* J **do**
$\vec{l}_J = pageRank(G_{JJ}, \vec{s}_J, \vec{v}_J)$;
end

2. Compute block transition matrix B and BlockRanks $\vec{b}$.

$B = L^T AS$
$\vec{b} = pageRank(B, \vec{v}_k, \vec{v}_k)$

3. Find an approximation $\vec{x}^{(0)}$ to the global PageRank vector $\vec{x}$ by weighting the local PageRanks of pages in block J by the BlockRank of J.

$\vec{x}^{(0)} = L\vec{b}$

4. Use this approximation as a start vector for a standard PageRank iteration.

$\vec{x}^{(0)} = pageRank(G, \vec{x}^{(0)}, \vec{v})$

Algorithm 8: BlockRank Algorithm

can then encode the personalization vector in the k-dimensional vector $\vec{v}_k$ (where k is the number of blocks (hosts) in the Web) that is a distribution over different hosts. As we alluded to earlier in Section 3.5, we can then compute a basis set of PageRank vectors that were generated using this block-level personalization scheme.

With this restriction on the random jump the surfer is allowed to make, the local PageRank vectors $\vec{l}_J$ will not change for different personalizations. In fact, since the local PageRank vectors $\vec{l}_J$ do not change for different personalizations, neither does the block matrix B. Only the BlockRank vector $\vec{b}$ will change for different personalizations. Therefore, we only need to recompute the BlockRank vector $\vec{b}$ for each block-personalization vector $\vec{v}_k$. Assuming you have already computed a generic PageRank vector once using the BlockRank algorithm, and have stored the block-transition matrix B, the personalized BlockRank algorithm is simply the last 3 steps of the generic BlockRank algorithm.

Inducing Random Jump Probabilities Over Pages

The Personalized BlockRank algorithm requires that the random surfer not have the option of jumping to a specific page when he teleports (he may only jump to the host). However, the last step in the BlockRank algorithm, in which we run the normal Power Method iterations over the page-level Web graph, requires a random jump probability distribution $\vec{v}$ over *pages*. Thus, we need to induce the probability $p(j)$ that the random surfer will jump to a page j if we know the probability $p(J)$ that he will jump to host J in which page j resides. We induce this probability as follows:

$$p(j) = p(J)p(j|J) \tag{5.1}$$

That is, the probability that the random surfer jumps to page j is the probability that he will jump to host J, times the probability of being at page j given that he is in host J.

Since the local PageRank vector $\vec{l}_J$ is the stationary probability distribution of pages within host J, $p(j|J)$ is given by the element of $\vec{l}_J$ corresponding to page j. Therefore, the elements L_{jJ} of the matrix L correspond to $L_{jJ} = p(j|J)$. Also, by

definition, the elements $(v_k)_J = p(J)$. Therefore, in matrix notation, Equation 5.1 can be written as $\vec{v} = L\vec{v}_k$.

Using "Better" Local PageRanks

If we have already computed the generic PageRank vector $\vec{x}$, we have even "better" local PageRank vectors than we began with. That is, we can normalize segments of $\vec{x}$ to form the normalized global PageRank segments $\vec{g}_J$ as described in Section 5.4.3. These scores are of course better estimates of the relative magnitudes of pages within the block than the local PageRank vectors $\vec{l}_J$, since they are derived from the generic PageRank vector for the full Web. So we can modify Personalized BlockRank as follows. Let us define the matrix H in a manner similar to the way we defined L, except using the normalized global PageRank segments $\vec{g}_J$ rather than the local PageRank vectors $\vec{l}_J$.

Again, we only need to compute H once. We define the matrix B_H to be similar to the matrix B as defined in Equation 5.4.3, but using H instead of L:

$$B_H = H^T AS \tag{5.2}$$

Using B_H (constructed from segments of the global PageRank vector) in place of B (constructed from the local PageRank vectors), we can proceed with Steps 2-4 of Algorithm 8.

5.4.5 Advantages of BlockRank

The BlockRank algorithm has four major advantages over the standard PageRank algorithm.

Advantage 1 A major speedup of our algorithm comes from caching effects. All of the host-blocks in our crawl are small enough so that each block graph fits in main memory, and the vector of ranks for the active block largely fits in the CPU cache. As the full graph does not fit entirely in main memory, the local PageRank iterations thus require less disk i/o then the global computations.

The full rank vectors do fit in main memory; however, using the sorted link structure[5] dramatically improves the memory access patterns to the rank vector. Indeed, if we use the sorted link structure, designed for BlockRank, as the input instead to the *standard* PageRank algorithm, the enhanced locality of reference to the rank vectors cuts the time needed for each iteration of the standard algorithm by over 1/2: from 6.5 minutes to 3.1 minutes for each iteration on DNR-LargeWeb!

Advantage 2 In the BlockRank algorithm, the local PageRank vectors for many blocks will converge quickly; the computations of those blocks may be terminated after only a few iterations. Thus, we can expend more computation on slowly converging blocks and less computation on faster converging blocks. Note for instance in Figure 5.9 that there is a wide range of rates of convergence for the blocks. In the standard PageRank algorithm, iterations operate on the whole graph; thus the convergence bottleneck is largely due to the slowest blocks. Much computation is wasted recomputing the PageRank of blocks whose local computation has already converged.

Advantage 3 The local PageRank computations in Step 1 of the BlockRank algorithm can be computed in a completely parallel or distributed fashion. That is, the local PageRanks for each block can be computed on a separate processor, or computer. The only communication required is that, at the end of Step 1, each computer should send their local PageRank vector $\vec{l_j}$ to a central computer that will compute the global PageRank vector. If our graph consists of n total pages, the net communication cost consists of $8n$ bytes (if using 8-byte double precision floating point values). Naive parallelization of the computation that does not exploit block structure would require a transfer of $8n$ bytes *after each iteration*, a significant penalty. Furthermore, the local PageRank computations can be pipelined with the Web crawl. That is, the local PageRank computation for a host can begin as a separate process as soon as the crawler finishes crawling the host. In this case, only the costs of Steps 2–4 of the BlockRank algorithm become rate-limiting.

[5]As in Section 5.4.2, this entails assigning document ids in lexicographic order of the url (with the components of the full hostname reversed).

Advantage 4 In several scenarios, the local PageRank computations (e.g., the results of Step 1) can be reused during future applications of the BlockRank algorithm. Consider for instance news sites such as cnn.com that are crawled more frequently then the general Web. In this case, after a crawl of cnn.com, if we wish to recompute the global PageRank vector, we can rerun the BlockRank algorithm, except that in Step 1 of our algorithm, only the local PageRanks for the cnn.com block need to be recomputed. The remaining local PageRanks will be unchanged, and can be reused in Steps 2–3. Recall from Section 5.4.4 that in the case of Personalized BlockRank, if we have a global PageRank vector available, we can avoid the local PageRank computations altogether.

5.4.6 Experimental Results

In this section, we investigate the speedup of BlockRank compared to the standard algorithm for computing PageRank. The speedup of our algorithm for typical scenarios comes from the first three advantages listed in Section 5.4.5. The speedups are due to less expensive iterations, as well as fewer total iterations. In the case of personalized computations, Advantage 4 helps us reduce redundancy across the computation of different personalized PageRank vectors.

We begin with the scenario in which PageRank is computed after the completion of the crawl; we assume that only Step 0 of the BlockRank algorithm is computed concurrently with the crawl. As mentioned in Advantage 1 from the previous section, simply the improved reference locality due to block structure, exposed by lexicographically sorting the link matrix, achieves a speedup of a factor of 2 in the time needed for each iteration of the standard PageRank algorithm. This speedup is completely independent of the value chosen for c, and does not affect the rate of convergence as measured in number of iterations required to reach a particular L_1 residual.

If instead of the standard PageRank algorithm (i.e., Power Method), we use the BlockRank algorithm on the block structured matrix, we gain the full benefit of Advantages 1 and 2; the blocks each fit in main memory, and many blocks converge more quickly than the convergence of the entire Web. We compare the wallclock

Step	Wallclock time
1	17m 11s
2	7m 40s
3	0m 4s
4	56m 24s
Total	**81m 19s**

Table 5.5: Running times for the individual steps of BlockRank for $c = 0.85$ in achieving a final residual of $< 10^{-3}$.

Algorithm	Wallclock time
Standard	180m 36s
Standard (using URL-sorted links)	87m 44s
BlockRank (no pipelining)	81m 19s
BlockRank (w/ pipelining)	57m 06s

Table 5.6: Wallclock running times for 4 algorithms for computing PageRank with $c = 0.85$ to a residual of less than 10^{-3}.

time it takes to compute PageRank using the BlockRank algorithm in this scenario, where local PageRank vectors are computed serially after the crawl is complete, with the wallclock time it takes to compute PageRank using the standard algorithm given in [61]. Table 5.5 gives the running times of the 4 steps of the BlockRank algorithm on the LargeWeb dataset. The first 3 rows of Table 5.6 give the wallclock running times for standard PageRank, standard PageRank using the URL-sorted link matrix, and the full BlockRank algorithm computed after the crawl. We see there is a small additional speedup for BlockRank on top of the previously described speedup. Subsequently, we will describe a scenario in which the costs of Steps 1–3 become largely irrelevant, leading to further effective speedups.

In this next scenario, we assume that the cost of Step 1 can be made negligible in one of two ways: the local PageRank vectors can be pipelined with the Web crawl, or they can be computed in parallel after the crawl. If the local PageRank vectors are computed as soon as possible (e.g., as soon as a host has been fully crawled), the majority of local PageRank vectors will have been computed by the time the crawl is finished. Similarly, if the local PageRank vectors are computed after the crawl, but in a distributed manner, using multiple processors (or machines)

	PageRank	BlockRank
STANFORD/BERKELEY	50	27
LARGEWEB	28	18

Table 5.7: Number of iterations needed to converge for standard PageRank and for BlockRank (to a tolerance of 10^{-4} for STANFORD/BERKELEY, and 10^{-3} for LARGEWEB).

to compute the PageRank vectors independently, the time it takes to compute the local PageRanks will be low compared to the standard PageRank computation. Thus, only the running time of Steps 2–4 of BlockRank will be relevant in computing net speedup. The contruction of B is the dominant cost of Step 2, but this too can be pipelined; Step 3 has negligible cost. Thus the speedup of BlockRank in this scenario is determined by the increased rate of convergence in Step 4 that comes from using the BlockRank approximation $\vec{x}^{(0)}$ as the start vector. We now take a closer look at the relative rates of convergence. In Figure 5.10(a), we show the convergence rate of standard PageRank, compared to the convergence of Step 4 of BlockRank on the STANFORD/BERKELEY dataset for a random jump probability $1 - c = 0.15$ (i.e., $c = 0.85$). Note that to achieve convergence to a residual of 10^{-4}, using the BlockRank start vector leads to a speedup of a factor of 2 on the STANFORD/BERKELEY dataset. The LARGEWEB dataset yielded an increase in convergence rate of 1.55. These results are summarized in Table 5.7. Combined with the first effect described above (from the sorted link structure), in this scenario, our algorithm yields a net speedup of over 3.

We next give results for the personalized BlockRank algorithm, in which we assume the generic PageRank vector has already been computed, so that there is no need to compute local PageRanks. We consider a random surfer who is a graduate student in linguistics at Stanford. When he teleports (with probability 0.85), he has a .8 probability of jumping to the linguistics host www-linguistics.stanford.edu, and a .2 probability of jumping to the main Stanford host www.stanford.edu. Figure 5.11 shows that the speedup of computing the Personalized PageRank for this surfer shows comparable speedup benefits to standard BlockRank. Note that the local PageRank vectors do not need to be computed at all for Personalized BlockRank, since the

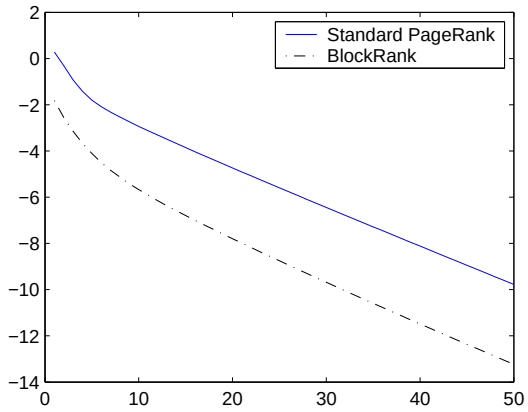


Figure 5.10: Convergence rates for standard PageRank (solid line) vs. BlockRank (dotted line). The x-axis is the number of iterations, and the y-axis is the log of the L_1-residual. STANFORD/BERKELEY data set; $c = 0.85$.

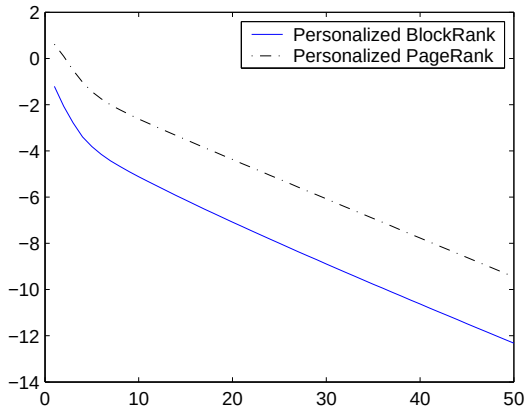


Figure 5.11: Convergence of Personalized PageRank computations using standard PageRank and Personalized BlockRank.

matrix H is formed from the already computed generic PageRank vector. Therefore, the running time characteristics of personalized BlockRank on the full Web are given by the "BlockRank w/ pipelining" results given earlier.

Chapter 6

Efficient Encodings for Ranking Vectors

6.1 Introduction[1]

In order for our system to utilize efficiently, at query-time, the topic-sensitive PageRank vectors that we computed offline, we need to compress them so that they can be stored in main memory. A detailed look at our query-processing subsystem shows why maintaining them in memory is important. As depicted in Figure 6.1, our keyword-search query-processing subsystem utilizes an inverted text index $\mathcal{I}$ and an auxiliary index $\mathcal{R}$ consisting of a set of topic-sensitive PageRank ranking vectors. For the moment, consider a simplified system with only one such vector containing the standard PageRank attribute. The index $\mathcal{I}$ contains information about the occurrences of terms in documents and is used to retrieve the set of document IDs for documents satisfying some query $\mathcal{Q}$. The index $\mathcal{R}$ is then consulted to retrieve the PageRank score for each of these candidate documents. Using the information retrieved from $\mathcal{I}$ and $\mathcal{R}$, a composite document score is generated for each candidate result, yielding a final ranked listing.

The inverted index $\mathcal{I}$ is constructed offline and provides the mapping $\{t \rightarrow f_{dt}\}$, where f_{dt} describes the occurrence of term t in document d. In the simplest case, f_{dt}

[1]This chapter covers work we first presented in [34, 36]

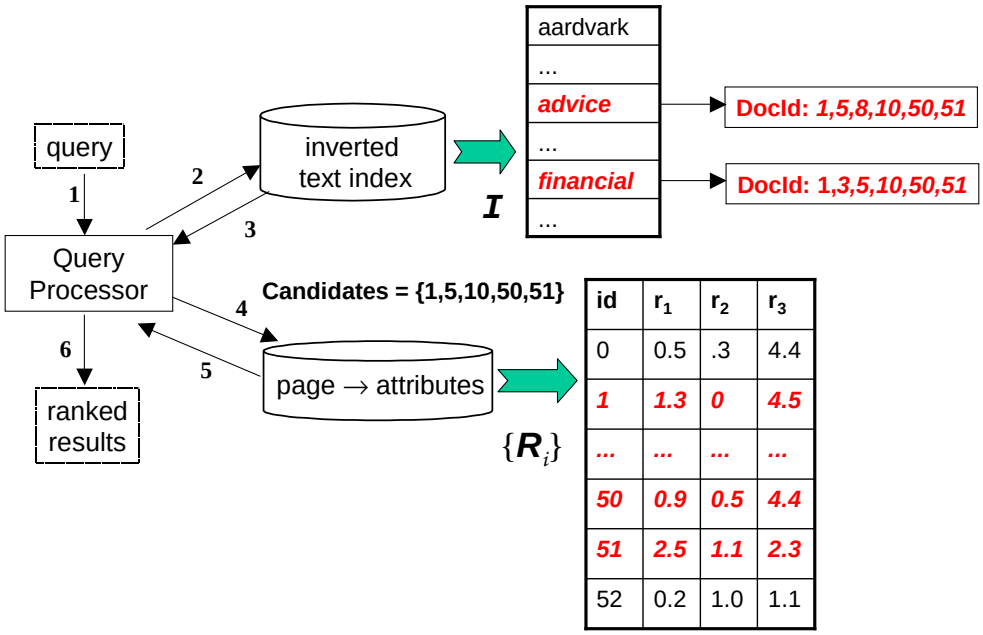


Figure 6.1: A simplified illustration of a search engine with a standard inverted text-index and 3 auxiliary numerical attributes for each document. Note that the number of random accesses to $\mathcal{I}$ is typically small, whereas the number of accesses to $\mathcal{R}$ is large. Our goal is to minimize the space needed for the data structure $\mathcal{R}$.

could be the within-document frequency of t. The number of random accesses to $\mathcal{I}$ needed to retrieve the necessary information for answering a query $\mathcal{Q}$ exactly equals the number of terms in the query, $|\mathcal{Q}|$. Because queries are typically small, consisting of only a few words, it is practical to keep the index $\mathcal{I}$ on-disk and perform $|\mathcal{Q}|$ seeks for answering each query.

The auxiliary index $\mathcal{R}$ is also constructed offline, and provides the mapping $\{d \rightarrow r_d\}$, where r_d is the popularity (e.g., PageRank) of document d. Note that in contrast to $\mathcal{I}$, the index $\mathcal{R}$ provides *per-document* information. The search system typically must access $\mathcal{R}$ once for *each* candidate document of the result set, which could potentially be very large. These random accesses would be prohibitively expensive, unless $\mathcal{R}$ can be kept entirely in main memory. Whereas the query length is the upper bound for the accesses to $\mathcal{I}$, the number of candidate results retrieved from $\mathcal{I}$ is the upper bound for accesses to $\mathcal{R}$. One way to reduce the number of random accesses required is to store the attribute values of $\mathcal{R}$ in $\mathcal{I}$ instead; e.g., create an

index $\mathcal{I}'$ that provides the mapping $\{t \rightarrow \{f_{dt}, r_d\}\}$. However, doing so would require replicating the value r_d once for each distinct term that appears in d, generally an unacceptable overhead, especially if more than one numeric property is used.

Much work has been done on compressing $\mathcal{I}$, although comparatively less attention has been paid to effective ways of compressing auxiliary numeric ranking vectors such as $\mathcal{R}$. The typical keyword-search system has only a few such auxiliary ranking vectors, such as the document lengths needed in computing the query-document cosine similarity [74] — and can be kept in main memory without much difficulty. However, our topic-sensitive PageRank based system requires consulting a *set* of auxiliary ranking vectors, so that much more consideration needs to be given to the encodings used for the attribute values.

Note that falling main-memory prices do not eliminate the need for efficient encodings; increasingly affordable disk storage is leading to rapidly growing Web-crawl repositories, in turn leading to larger sets of documents that need to be indexed. Furthermore, to support large numbers of simultaneous queries, search engines must maintain tens or hundreds of replicas of search indexes on independent machines, exacerbating the need for efficiently encoded indexes. Utilizing a rich set of per-document numeric ranking attributes for growing crawl repositories and growing numbers of users thus continues to require efficient encoding schemes.

In Section 6.2, we briefly review scalar quantization, which provides the framework for our approach, and introduce new distortion criteria for measuring quantizer performance. In Section 6.3, we discuss fixed-length encoding schemes and analyze their performance using a traditional numerical distortion measure. In Section 6.4, we investigate in detail distortion measures more appropriate for the case of search ranking functions, and analyze the performance of various quantization strategies both analytically and empirically. In Section 6.5, we extend our approach to consider variable-length encoding schemes and discuss their performance. We review related work in Section 6.6 and conclude in Section 6.7.

6.2 Scalar Quantization

6.2.1 Quantization Rules

An excellent introduction to quantization can be found in [28]; we give only a brief review here. Let $\mathcal{C} \subset \mathbb{R}$; for our work, assume $\mathcal{C}$ is finite. A quantizer is a function $q(x) : \mathbb{R} \rightarrow \mathcal{C}$ that partitions $\mathbb{R}$ into a set $\mathcal{S}$ of intervals and maps values in the same interval to some common *reproduction value* in $\mathcal{C}$. In other words, $q(x)$ maps real values to some approximation of the value. Let $n = |\mathcal{C}|$. As the values in $\mathcal{C}$ can be indexed from 0 to $n-1$, one way to compactly represent values in the range of $q(x)$ is with fixed-length codes of length $l = \lceil log_2 n \rceil$ bits, in conjunction with a codebook mapping the fixed-length codes to the corresponding reproduction value. Let $\hat{x} = q(x)$. Given the sequence $\{a_i\}$ of real numbers as input, a compression algorithm based on fixed-length scalar quantization would output the sequence of l-bit codewords $\hat{a_i}$, along with the codebook mapping each *distinct* codeword to its corresponding reproduction value. The error that results from quantizing a particular input value x to the reproduction value $\hat{x}$ is typically quantified by a *distortion* measure. We consider distortion measures in Section 6.2.2.

The simplest fixed-length encoding simply partitions the domain of possible values into n cells of uniform width using a uniform quantizer u_n, where n is typically chosen to be a power of 2. A more complex quantizer could use a nonuniform partition to lower the distortion. Alternatively, instead of using nonuniform partitions, the input values can be transformed with a nonlinear function $G(x)$, called a compressor, then uniformly quantized using u_n. The inverse function $G^{-1}(x)$ can be used for reconstructing an approximation to the original value. Such a quantizer $G^{-1}(u_n(G(x)))$ is called a *compander*[2]; it is known that any fixed-length, nonuniform quantizer can be implemented by an equivalent compander [28]. For simplicity, unless otherwise noted, we will define quantization strategies as companders.

Quantizers can also make use of variable-length codes for the elements in set $\mathcal{C}$. If shorter codewords are assigned to the elements in $\mathcal{C}$ that more frequently correspond

[2]Short for *compressor, expander*. Note that companders save on the need for explicit codebooks, as the partitioning of the domain is uniform. $G^{-1}(x)$ takes the place of the codebook.

to values in the input data being compressed, the *average* codeword length can be reduced. The simplest scheme would use a Huffman code for $\mathcal{C}$, using the known or estimated frequency of the elements in $\mathcal{C}$ to generate the optimal Huffman codes.

As a sidenote, we mention that one possible way to encode the ranks for documents is to sort the documents by rank, and then assign each document a codeword corresponding to the ordinal rank of that document. Since there are hundreds of millions of documents, each codeword still requires at least 28 bits, so that this scheme does not solve the basic problem of how best to efficiently encode ranking attributes (for more than one such attribute).[3]

6.2.2 Measuring Distortion

The scalar quantization literature in general considers the loss in numerical precision when comparing the expected distortion of quantization schemes. For instance, the most commonly used measure of distortion for a value is the squared error:

$$d(x, q(x)) = (x - q(x))^2 \tag{6.1}$$

The inaccuracy of a particular quantization function q for a particular set of input data is then the mean distortion, denoted $D(q)$. If $d(x, q(x))$ is the squared error as defined above, then $D(q)$ is referred to as the *mean squared error*, or MSE.

However, in the case of document ranking, the numerical error of the quantized attribute values themselves are not as important as the effect of quantization on the *rank order* induced by these attributes over the results to a search query. In our work, we show that distortion measures based on induced rankings of search-query results lead to different choices for optimality.

Assume each document in the corpus has k associated numerical ranking attributes. Note that some of these attributes, such as PageRank, are precomputed and stored, and some are query-specific and hence generated at query-time. As the goal of quantization is to reduce space requirements of the precomputed indexes, it is

[3]This scheme can make certain quantization rules easier to implement, but does not eliminate the need to construct efficient quantizers.

used only on the precomputed attributes. The attributes can be used in one of two ways to rank a set of documents that are candidate results to some search query.[4] The scenarios we consider are:

1. Each of the k attributes can be used separately to rank the candidate result documents to generate k intermediate rank orders, which are then *aggregated* to generate a final rank order over the candidates [23].

2. The values for the k attributes for the documents can be combined numerically to form a composite score, which is then used to rank the set of candidate documents.

Under Scenario 1, quantization has a very simple effect on the intermediate rank orders. Quantization can map similar values to the same cell, but can never swap the relative order of two values; for any two values x and y, and any quantizer q, we know that $x < y \Rightarrow q(x) \leq q(y)$. Thus, an intermediate rank order using a quantized version of an attribute differs from the intermediate rank order using the original attribute only through the introduction of false ties.[5] This property suggests the following distortion measure. Let the distortion of a quantizer on a particular attribute, for a particular candidate result set of size m, be measured as the sum of squares of the number of candidate documents mapped to the same cell, normalized so that the maximum distortion is 1. Assuming the original values for the attribute were distinct, this distortion is closely related to the fraction of document pairs in the result set that are falsely tied. More formally, let $\mathcal{S}$ be the query-result set, with $m = |\mathcal{S}|$, and let X_i be the number of documents in $\mathcal{S}$ mapped to cell i for the attribute under consideration. The distortion of an n-cell quantizer on the set $\mathcal{S}$ is given by:

$$Distortion(q_j, \mathcal{S}) = \frac{1}{m^2} \sum_{0}^{n-1} X_i^2 \tag{6.2}$$

[4]E.g., the candidate result set might consist of documents that contain all of the query terms.

[5]The final rankings, after rank aggregation, may of course differ in more complex ways, depending on how the aggregation is done. If in such a case, measuring the distortion of final rankings is desired, the distortion measure of Scenario 2 is more appropriate.

We refer to this distortion measure as TDist. We can evaluate the relative performance of different quantizers based on the expectation (or average over some test query set) of the above distortion. Different models for computing this expectation are given in Section 6.4. The empirical performance of different quantizers over a test set of queries on this distortion measure are presented in Section 6.4.5.

Under Scenario 2, we cannot use the above distortion measure. The error can no longer be measured solely through artificial ties. In the final rankings induced by the composite score, the relative ordering of documents can be different. To measure the distortion in this case, we rely on the KDist measure we defined earlier in Appendix 4.A. Consider two partially ordered lists of URLs, τ and τ_q, each of length m, corresponding to rankings induced by exact and approximate composite scores, resp. As our distortion measure, we use $\text{KDist}(\tau, \tau_q)$; recall that this value is the probability that τ and τ_q disagree on the relative ordering of a randomly selected pair of distinct nodes (u, v). We present the empirical performance of different quantizers on the above distortion measure in Section 6.4.5.

6.3 Fixed-Length Encoding Schemes

In this section, we discuss fixed-length encoding schemes, describe the optimal encoding under the MSE distortion measure, and give the empirical MSE-performance of various fixed-length encoding schemes. The "best" fixed-length quantizer q can be chosen by answering the following three questions:

1. What is the appropriate measure of distortion $D(q)$ for the application?
2. How many cells should the partition have? In other words, what is the appropriate choice for n, noting that (a) the codeword length is given by $\lceil log_2 n \rceil$, and (b) smaller n will lead to higher distortion.
3. For a particular n, what compressor function $G(x)$ should be used to minimize distortion?

Answering Question 2 is simply a matter of choosing a codeword length that will allow the encoded ranking vector to fit in the available memory. If we answer Question 1 by

choosing the MSE, then results from the quantization literature [28, 63], let us choose the optimal compressor function based on the distribution of the input values, leading to the answer for Question 3. We begin with this case (i.e., where $D(q)$ is the MSE), and then discuss the use of more appropriate distortion measures in Section 6.4.

The optimal compressor function $G(x)$ (i.e., the $G(x)$ which will minimize the quantization MSE) is determined by the probability density function (pdf) of the input data, $p(y)$. In particular, the optimal compressor is given by the following [63]:

$$G(x) = c \cdot \int_{-\infty}^{x} p(y)^{1/3} dy \tag{6.3}$$

Fortunately, the entire ranking vector that we would like to encode is available, so we can determine $p(y)$ easily. In Section 6.3.1, we look at the distribution of values of PageRank vectors in our dataset, and in Section 6.3.2, we discuss the MSE performance of 6 fixed-length coding schemes.

6.3.1 Data Distribution

To compute the right hand side of Equation 6.3, we examine the relative frequency distribution $p(y)$ of the values of the PageRank ranking vectors that we want to encode.

We used as our dataset the Stanford WebBase crawl repository of 120 million pages, containing a total of 360 million distinct URLs. This latter count includes pages that were linked to from crawled pages, but were not themselves crawled. Note that using a standard 4-byte floating point representation, the PageRank vector for these 360 million pages requires 1.34GB of space. Figure 6.2 shows the distribution of the standard PageRank values for this dataset on a log-log plot. In plotting the distribution, we used logarithmic binning, with the counts normalized by the bin width. In other words, when computing the relative counts shown on the y-axis, we used bins of equal width on a logarithmic scale, and divided the counts by the actual width of the bin. This technique was needed, as the data is very sparse for high rank

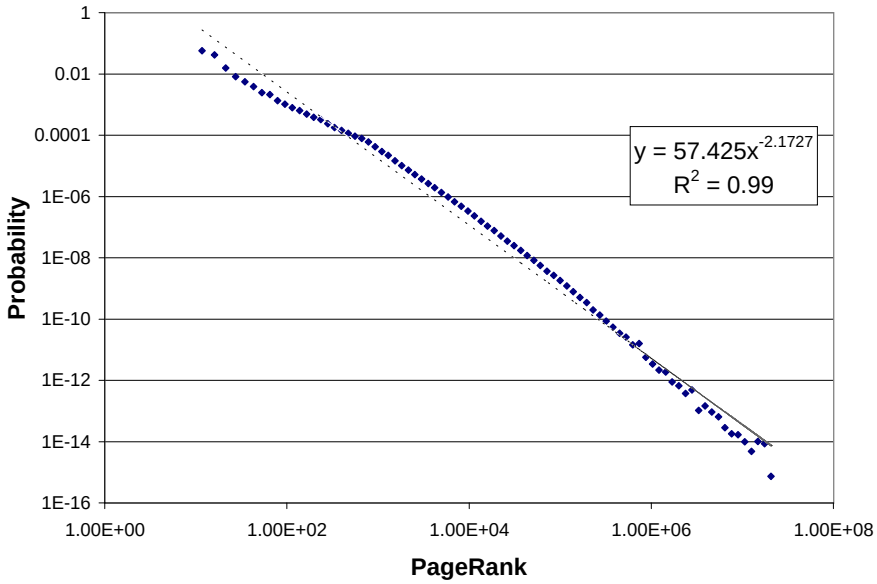


Figure 6.2: PageRank distribution on a log-log scale. The crawl graph included 360M URLs generated from a 120M page repository. The best-fit power-law curve is shown, with a slope close to -2.1, in agreement with prior findings.

values.[6] The distribution appears to follow a power-law with exponent close to 2.17, similar to the findings of Pandurangan et al. [62] on a different dataset. They also provide a graph-generation model that explains this observed property of the Web.

The Topic-Sensitive PageRank vectors we constructed following the methodology proposed in Chapter 3 behave similarly. For instance, the values for the PageRank vector generated with respect to the COMPUTERS topic follow the distribution shown in Figure 6.3; the other topic-specific PageRank scores we measured also follow a similar distribution, but are not shown here. Note that the power-law fit is not quite as close, with the slope steepening noticeably in the tail. The best-fit power-law exponents for the topic-specific PageRank distributions all ranged between 1.7 and 1.8. Because developing efficient encodings for the topic-specific rank vectors is analogous to developing encodings for the standard PageRank vector, the remaining discussion focuses solely on encodings for the standard PageRank vector.

When computing the optimal compressor function for the standard PageRank

[6] A similar approach is used in [64].

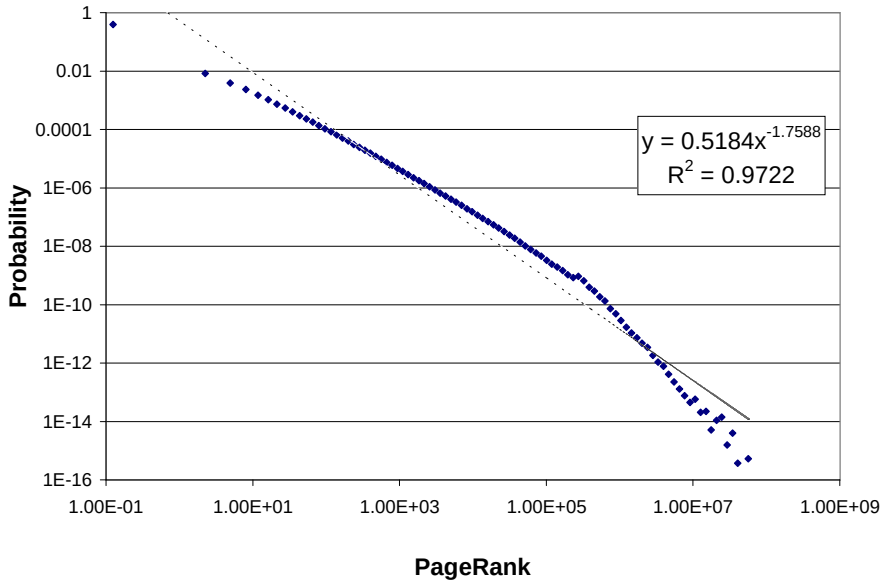


Figure 6.3: COMPUTERS-biased PageRank distribution on a log-log scale. The slope is close to -1.8, less steep than the ordinary PageRank distribution.

vector, for minimizing the MSE, we compute Equation 6.3 with the pdf $p(y) = k \cdot y^{-2.17}, y > y_{min}$.

6.3.2 MSE Performance of Fixed-Length Schemes

In this section, we compare the performance of various fixed-length encoding schemes using the mean-squared-error (MSE) measure. Except for the **equal_depth** strategy, the quantizers are implemented as companders. A summary of the strategies we consider is given in Table 6.1. To illustrate the behavior of the quantization strategies on the PageRank values for the 360 million URLs, Figure 6.4[7] shows the relative number of input points mapped to each cell (i.e., the depth of each cell) for 6 different strategies when using 256 cells. The y-axis is a log scale, to facilitate comparison. The figure depicts how the various compressor functions transform the input data, whose distribution was earlier shown in Figure 6.2. Because of PageRank's power-law distribution, we see that for the **linear** (i.e., uniform-width) strategy most cells are

[7]For clarity, in all of the graphs that follow, the order of the entries in the graph legend reflects the relative position of the corresponding curves in the graph.

Table 6.1: A description of 6 quantization strategies we compare.

Strategy	Description
linear	A uniform quantizer (partition uses cells of equal width)
sqrt	Compander with $G(x) \propto \sqrt{x}$
log	Compander with $G(x) \propto \log x$
mse_optimal	Compander with $G(x) \propto x^{-2.17}$
approx_eq_depth	Compander that approximates equal depth partitions, as described in Section 6.4.3
eq_depth	Quantizer where partition assigns an equal number of points to each cell

empty. The other 4 strategies use nonuniform partitions to divide up the range of possible PageRank values.

To compare the quantization strategies in the traditional (numeric) way, we computed the MSE for each strategy for encoding the standard PageRank vector. We varied the number of cells used from 2^4 to 2^{24}; i.e., the number of bits necessary for a fixed-length code varied from 4 to 24 bits per value. Figure 6.5 graphs the MSE vs. code-length for each of the strategies. We see that **mse_optimal** performs the best, as expected, with **log** and **sqrt** not far behind. We will see in the following section, however, that if we use a distortion measure based on the induced rankings of query results, rather than the MSE, the choice of optimal strategy differs.

6.4 Optimizing for Rank-Based Distortion Measures

We now discuss how we to choose the optimal quantization rule in the context of search ranking, under various distortion measures and probabilistic models for the keyword-search task. In general, unless both the search model and distortion measure are fairly simple, analytically deriving the optimal quantization rule becomes complex. We derive optimal quantization rules for simple cases, and rely solely on experimental data for more complex cases.

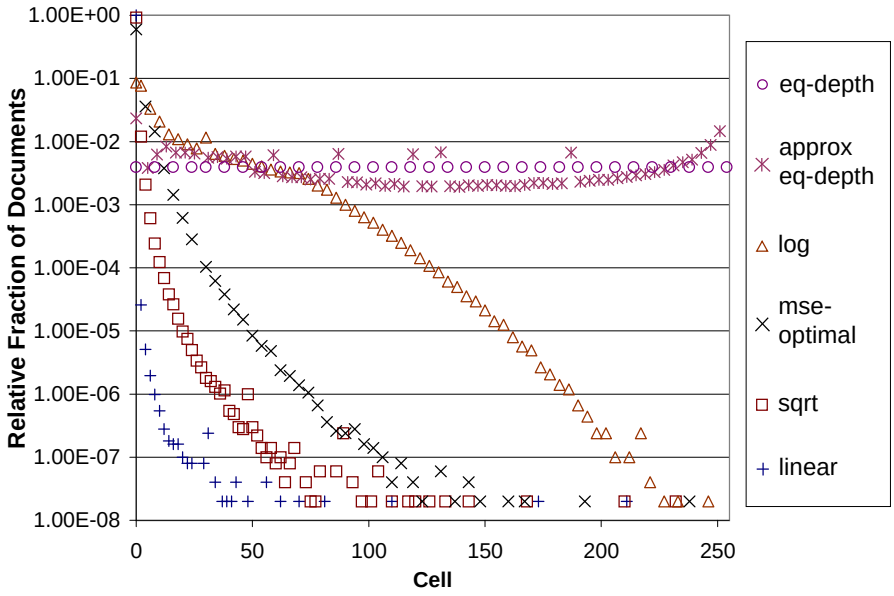


Figure 6.4: Relative cell counts for the various strategies for 8-bit codes (i.e., 256 cells). The y-axis gives the fraction of the values in the input mapped to each cell.

In Section 6.4.1, we introduce a simplified model of the keyword search process. We analytically derive the optimal quantization strategy for this model in Section 6.4.2, and then extend the derivation to richer models of search. Section 6.4.3 describes a technique to approximate the optimal strategy using a simple compander. In Section 6.4.4, we present empirical results describing the distribution of our data that provides justification for our simplified models. Section 6.4.5 presents experimental results illustrating the performance of the quantization strategies under various ranking models and corresponding distortion measures.

6.4.1 Retrieval and Ranking Model

We begin with a greatly simplified model of keyword search to allow for the analysis of the effects of quantization on query-result rankings, and later discuss extensions.

The first part of the model describes the retrieval of the candidate documents for a query. Let $\mathcal{D}$ be the set of documents in the Web-crawl repository. **Retrieve**$(\mathcal{D}, \mathcal{Q})$ is defined as the operation that returns the set $\mathcal{S} \subset \mathcal{D}$ consisting of documents that

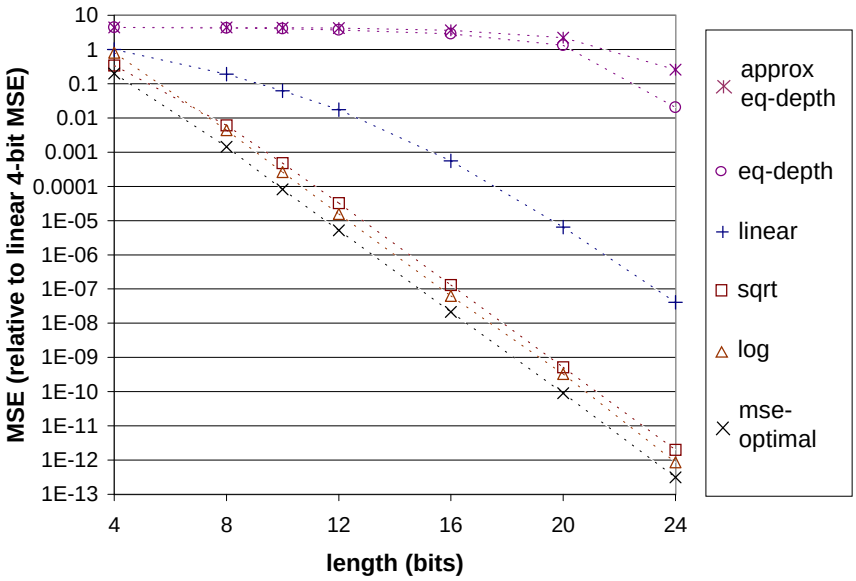


Figure 6.5: The MSE of 6 different strategies, for codeword lengths varying from 4 to 24. The MSE axis is on a log scale, and is normalized so that the MSE of the 4-bit linear compander is 1.

satisfy the query $\mathcal{Q}$.[8] For simplicity, we model the operation **Retrieve**$(\mathcal{D}, \mathcal{Q})$ as generating a random sample of size M from $\mathcal{D}$, with each element of $\mathcal{D}$ having an equal probability of appearing in the result set.

The second part of the model describes the ranking of the documents. Consider a single auxiliary ranking vector that is used to rank the documents in $\mathcal{S}$; e.g., assume that the candidate results will be ranked solely by their PageRank, ignoring any additional information available. Also assume that all full-precision PageRank values for the candidate documents are distinct.[9] The full-precision PageRank values $\vec{x}$ induce a total ordering over the set $\mathcal{S}$. If for each document d we use the quantized PageRank value, $q(r_d)$, then a weak ordering is induced instead. In other words, the relative order of the documents in $\mathcal{S}$ are preserved except for false ties between documents with PageRank values mapped to the same quantizer cell.

The third part of the model is the distortion measure used to judge the rank-order inaccuracy caused by quantization. In the simplified scenario being developed, this measure consists of penalizing the false ties, as described in Scenario 1 of Section 6.2.2.

[8] E.g., $\mathcal{S}$ could be the set of documents that contain all of the query terms in $\mathcal{Q}$.

[9] This assumption fails to hold only for values close to the minimum.

6.4.2 Derivation of Optimal Quantizers

For the simple model just discussed, we now derive the optimal quantizer. In particular, consider the case where **Retrieve**$(\mathcal{Q}, \mathcal{D})$ samples M documents from a repository $\mathcal{D}$ uniformly at random, without replacement. Let X_i be the number of query results mapped to cell i.

The number of documents returned by **Retrieve**$(\mathcal{Q}, \mathcal{D})$ will be different for different queries; i.e., M is a random variable. For now, consider the case where $M = m$ for some constant m. As described in Section 6.2.2, we let the distortion of a particular result of length m be measured by the sum of squares of the number of points in the same cell, normalized so that the maximum distortion is 1; in other words, we measure the distortion of the results $\mathcal{S}$, where $|\mathcal{S}| = m$, using a quantizer with n cells as:

$$Distortion_m(q_j) = \frac{1}{m^2} \sum_{0}^{n-1} X_i^2 \tag{6.4}$$

We can treat $\mathcal{D}$, the documents in the corpus, as a multi-type population, with n types. The type of each document is simply the cell it is mapped to by the quantizer q_j. Let N_i represent the number of points in the input data set[10] that the quantizer maps to cell i (i.e., the count of each type), and let N be the total number of input values (i.e., $N = \sum N_i$). Because the operation **Retrieve**$(\mathcal{D}, \mathcal{Q})$ samples from the population $\mathcal{D}$ uniformly at random, without replacement, $< X_0, \ldots, X_{n-1} >$ follows the multivariate hypergeometric distribution, with parameters m and $< N_0, \ldots, N_{n-1} >$. We assume that $|\mathcal{D}| \gg |\mathcal{S}|$, so that the multinomial distribution (i.e., the distribution that would arise if **Retrieve** sampled *with* replacement), with parameters m and $< \frac{N_0}{N}, \ldots, \frac{N_{m-1}}{N} >$, is a reasonable approximation.[11] The task of finding the optimal n-cell quantizer is reduced to choosing cell depths which minimize the expectation of Equation 6.4. Note that linearity of expectation allows us to consider each X_i

[10] Namely, the PageRank vector we are compressing.

[11] The approximation has no impact on the final solution; see Appendix 6.A for the full derivation using the multivariate hypergeometric distribution

separately, even though they are not independent:

$$E[Distortion_m] = \frac{1}{m^2} E[\sum X_i^2] \tag{6.5}$$

$$= \frac{1}{m^2} \sum E[X_i^2] \tag{6.6}$$

Since each X_i follows a binomial distribution, $E[X_i^2]$ is easy to find. Letting $p_i = \frac{N_i}{N}$, and using the known mean and variance of binomial random variables [29], we see that

$$E[X_i] = mp_i \tag{6.7}$$

$$var[X_i] \equiv E[X_i^2] - E[X_i]^2 \tag{6.8}$$

$$var[X_i] = mp_i(1 - p_i) \tag{6.9}$$

$$E[X_i^2] = mp_i(1 - p_i) + (mp_i)^2 = mp_i + m(m-1)p_i^2 \tag{6.10}$$

So we need to find the p_i that minimizes

$$E[Distortion_m] = \frac{1}{m^2} \sum E[X_i^2] \tag{6.11}$$

$$= \frac{1}{m^2} \sum_i \left(mp_i + m(m-1)p_i^2\right) \tag{6.12}$$

$$= \frac{1}{m} + \frac{m-1}{m} \sum_i p_i^2 \tag{6.13}$$

The above is equivalent to minimizing $\sum_i p_i^2$ subject to the constraint $\sum_i p_i = 1$. Lagrange multipliers can be used to show that the optimal solution is given by

$$p_i^* = \frac{1}{n} \tag{6.14}$$

$$N_i^* = Np_i^* = \frac{N}{n} \tag{6.15}$$

In other words, an equal-depth partition scheme that places equal numbers of points in each cell minimizes the expected distortion of the query results for the TDist distortion measure.

The above considered the case where M, the number of results, was fixed to some constant m. However, different queries have different numbers of results, so that M is a random variable. However since Equation 6.15 is independent of m, the optimal solution in the case where M varies is also given by Equation 6.15.

We now discuss several extensions to make our model of the operation **Retrieve** more realistic. Consider the case where the candidate query-results are first pruned based on a threshold for their cosine similarity to the query, then ranked purely by the quantized PageRank $q(r_d)$. The intuition behind this model is that the ranking function first chooses a set of documents thought to be relevant to the query, and then ranks these relevant candidates by their popularity. Our experiments showed virtually no correlation between the PageRank of a document, and its cosine similarity to queries.[12] Thus, since the *pruned* candidate set is expected to follow the same distribution as the *raw* candidate set, the optimal solution is unchanged in this new model.

A second extension to the model is to make the random sampling nonuniform. In other words, each document can have a different probability of being chosen as a candidate result. In this case, the hypergeometric distribution no longer applies since different objects of a given type have different probabilities of being chosen.

We could assume that the result set $\mathcal{S}$ is constructed by a sequence of m multinomial trials (i.e., sampling with replacement). Let $p(d)$ be the probability of document d being chosen during a trial. Let $p(cell_i) = \sum_{d_j \in cell_i} p(d_j)$. Then the random vector $< X_0, \ldots, X_{n-1} >$ follows the multinomial distribution with parameters m and $< p(cell_0), \ldots, p(cell_{n-1}) >$. The previous multinomial assumption holds if in addition to the requirement $|\mathcal{D}| \gg |\mathcal{S}|$, we also stipulate that no document dominates the probability mass of its cell. If $p(d)$ is extremely nonuniform among documents with similar values of the attribute being quantized, then sampling *with* replacement is no longer a good approximation to sampling *without* replacement. If the multinomial approximation does hold, then a derivation, similar to the above, shows that an *equiprobable* partition is optimal.

[12]The exact correlations are not given here, but were all close to zero. This result is expected, since PageRank is a purely link-based, *query-independent* estimate of page importance.

In other words, instead of making the depths of all cells constant, we make the probability mass assigned to each cell constant:

$$\sum_{d_j \in cell_i} p(d_j) = \frac{1}{n} \tag{6.16}$$

A third extension to the model we consider is the following. Assume that in addition to cosine pruning (which does not alter the distribution of ranking values in $\mathcal{S}$), we restrict the result set to the top k ranked documents. In this case, clearly the distribution of the pruned result set, $\mathcal{S}$, of size k, can no longer be modeled as a uniform random sample of $\mathcal{D}$. However, in this case, we can make use of the previous extension. In particular, the pruning to the top k results can be modeled by setting $p(d)$ to be some function of r_d, the PageRank of d.

In the general case, where the final rankings are generated using arbitrary ranking functions that numerically combine the scores from several ranking vectors, developing a probabilistic model for analytically deriving a solution becomes difficult; for such cases, we currently rely on empirical results, measuring the average distortion across a large number of sample queries.

6.4.3 Approximating Equal-Depth Partitioning

Using an equal-depth partition, although optimal for the TDist distortion measure, could lead to additional overhead. In the encoding phase, the true equal-depth scheme would require a binary search through the interval endpoints to determine the appropriate cell for each input value. Since the encoding step is performed offline, the cost is acceptable. However, in the decoding step, if the reproduction value for a particular cell is needed, a true equal-depth partition scheme requires a codebook that maps from cells to cell centroids, leading to additional space as well as processing costs. We show how we can approximate an equal-depth partition by using a simple compander, with a compressor function derived from the distribution of the underlying data, thus eliminating both the need for binary searches when encoding, and the need for a codebook at runtime.

If the input data values were distributed uniformly, we would intuitively expect

that a uniform partition would be similar to an equal-depth partition. We confirm this intuition as follows. Let N be the total number of points, and let N_i be the number of points that fall in cell i, for a quantizer with n cells. If the input points are distributed uniformly at random, then clearly each N_i follows the binomial distribution with parameters $(N, \frac{1}{n})$. Thus, the expected number of points in each cell is simply $\mu = E[N_i] = \frac{N}{n}$. The probability that N_i falls within a tight range of this expectation is high for large N, with $n \ll N$. For instance, for $N = 10^8, n = 10^6$, and using the normal approximation for N_i, we get that $Pr[.8\mu \leq N_i \leq 1.2\mu] \cong 0.95$.

Note, however, that the input data is *not* uniform. In particular, as we saw in Section 6.3.1, PageRank closely follows a power-law distribution. However, we can devise a compressor function $G_{eq}(x)$ that transforms the data to follow a uniform distribution, which we can then uniformly quantize, thus approximating an equal-depth quantizer. Let X be a random variable following the power-law distribution, with exponent 2.17; i.e., the pdf $f(x)$ for X is $f(x) = kx^{-2.17}$. Equivalently, if x_{min} is the minimum possible rank, and x_{max} is the maximum possible rank, the cumulative distribution function (cdf) [13] is $F(x) = \frac{k}{1.17}(x_{min}^{-1.17} - x^{-1.17})$. The normalization constant c is chosen so that $F(x_{max}) = 1$. We would like to find a function $G_{eq}(x)$ such that $G_{eq}(X)$ corresponds to a uniform distribution, i.e., a $G_{eq}(x)$ such that

$$Pr[G_{eq}(X) \leq y] = y \tag{6.17}$$

But it is easy to see that in fact, $F(x)$ itself is such a function, since the cumulative distribution of $F(X)$ is:[14]

$$Pr[F(X) \leq x] = Pr[X \leq F^{-1}(x)] = F(F^{-1}(x)) = x \tag{6.18}$$

Thus, a function $G_{eq}(x)$ that will transform the PageRank data to uniformly distributed data is the cdf $F(x)$, assuming that the PageRank distribution is a close fit for the power-law. This transformation allows us to eliminate the explicit codebook, instead using $G_{eq}(x)$ and $G_{eq}^{-1}(x)$ as the compressor and expander functions, resp., to

[13] The cdf is simply $\int_{x_{min}}^{x} pdf(y)dy$

[14] Note that $\forall_x f(x) > 0$ implies that $F(x)$ is strictly increasing, and thus invertible.

approximate an equal-depth partition. The empirical distribution of $G_{eq}(X)$, shown in Figure 6.4 as the series **approx_equal_depth**, does indeed appear to be roughly uniform. The results that will be described in Section 6.4.5 show that in practice, this approximation leads to performance similar to that of exact equal-depth partitioning.

6.4.4 Data Distribution: Corpus vs. Query Results

In Section 6.4.2, for deriving the optimal quantizer for the TDist distortion measure (Equation 6.5), we assumed that the operation **Retrieve**$(\mathcal{Q}, \mathcal{D})$ could be modeled as a uniform random sample of $\mathcal{D}$. We now present empirical data showing that this assumption is reasonable. We plotted the PageRank distribution of the raw query results for each of 86 test queries; in every case, the PageRank distribution closely matched the distribution of PageRank values in the corpus $\mathcal{D}$ as a whole. We display the distribution of PageRank values for the results of two representative queries in Figure 6.6.

The distributions were not an *exact* match, however, leading to the possibility that equiprobable and equal-depth partitions will not behave identically. We randomly partitioned our set of test queries into two halves. Using the first set, we measured the distribution of PageRank values of documents in the results for the queries. Figure 6.7 shows that this distribution deviates slightly from the distribution of PageRank values in the corpus as a whole; the distribution of PageRank values when restricted to documents that appear as raw candidate results seems to follow a power-law distribution with exponent close to 2.0. In other words, under our simplified model of the operation **Retrieve**, higher rank documents have a slightly higher probability of appearing in raw query results.[15]

[15]By "raw query results," we simply mean the set of documents containing all of the query terms for some query.

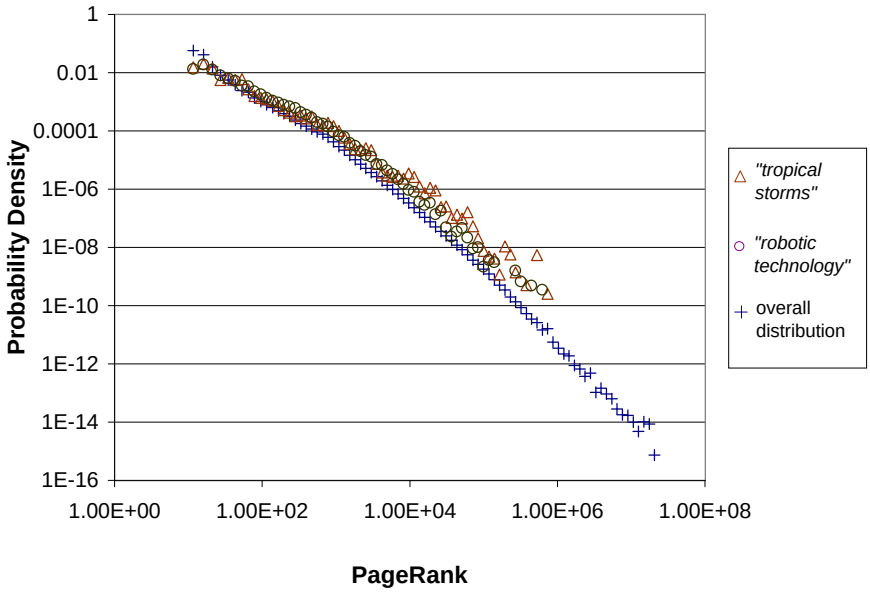


Figure 6.6: Comparison of the PageRank distribution for the corpus as a whole, to the set of pages containing the query terms "tropical storms," and the set of pages containing the query terms "robotic technology." As expected, the PageRank distribution for the raw conjunctive query results is close to the distribution on the overall corpus.

6.4.5 Empirical Performance Under Rank-Based Distortion Measures

We now explore the empirical performance of various quantization schemes on sample query results. Our test set of 86 queries consisted of 36 queries compiled from previous papers and 50 queries created using the titles from the TREC-8 topics 401-450 [59]. Using a text index for our repository, we retrieved, for each query, the URLs for all pages that contain all of the words in the query.

Figure 6.8 plots the average (over the 86 query results) of the distortion for the 6 strategies when using the TDist distortion measure. We see that as expected, the **equal_depth** strategy performs the best for all codelengths. Also note that the **approx_equal_depth** and **log** strategies perform similarly. The **mse_optimal** strategy, which was optimal when using the MSE distortion criteria, is no longer the optimal choice — this result signifies the need to consider appropriate notions of

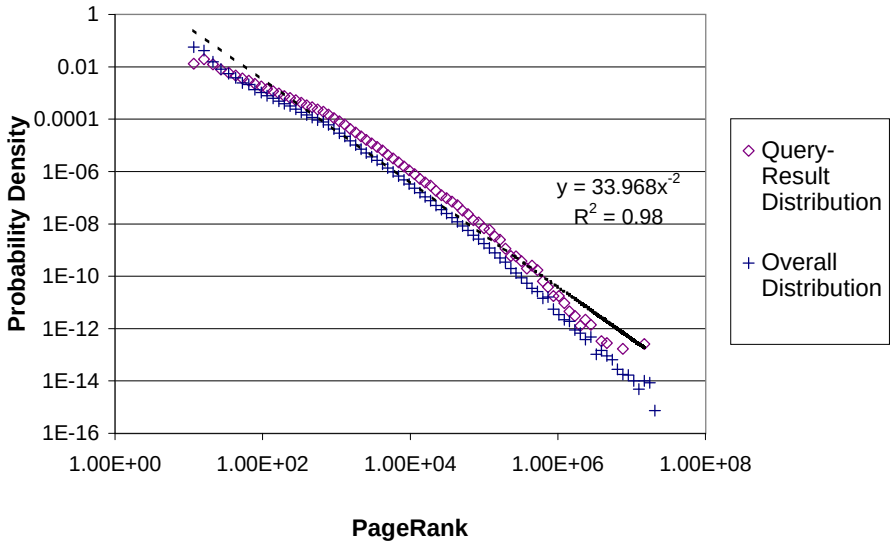


Figure 6.7: Comparison of the PageRank distribution for the repository as a whole, to the distribution for the set of results to 43 of the test queries.

distortion when choosing quantization strategies for search rankings.

In Section 6.4.2, we argued that for the TDist distortion measure, for a retrieval model in which documents in the corpus have different probabilities of appearing in the results, an equiprobable, rather than an equal-depth, partition is superior. As mentioned in Section 6.4.4, we noticed a slight correlation between the PageRank and the probability of appearing in the raw candidate result set. To test the performance of the **equal_prob** strategy, we implemented the compander described in Section 6.4.3, replacing the pdf with $f(x) = kx^{-2.0}$. On the subset of test queries that were not used in estimating the power-law exponent, we measured the performance, using the TDist distortion measure, of this approximation to an equiprobable partition. The results are shown in Figure 6.9, and demonstrate that the (approximate) equiprobable scheme improves upon the (approximate) equal-depth scheme. However, the true equal-depth scheme still performs the best by a small margin.

Figure 6.10 plots the average TDist distortion when the query results are first pruned to include only the top 100 documents based on the pure cosine similarity to the search query, then ranked using only the quantized PageRank value. We see that as expected, the **equal_depth** strategy performs the best for all codelengths. Also

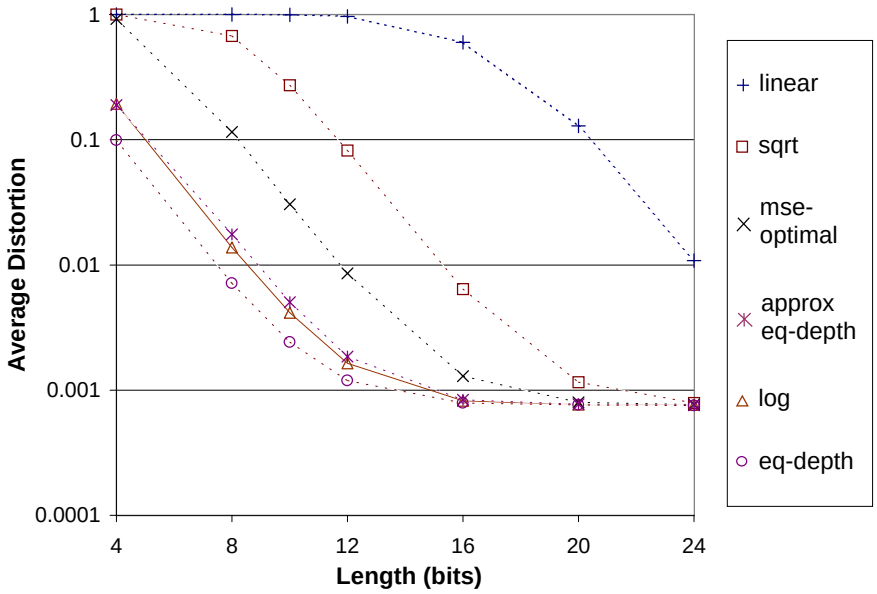


Figure 6.8: The average distortion for the various strategies when using the TDist distortion measure (Equation 6.5) over the full lists of query results, for 86 test queries.

note that the **log** and **approx_equal_depth** strategies also perform similarly. Notably, the **mse_optimal** strategy, which was optimal when using the MSE distortion criteria, is no longer the optimal choice – this result signifies the need to consider appropriate notions of distortion when choosing quantization strategies for search rankings.

As expected, the results match the results of Figure 6.8; since cosine similarity is uncorrelated with PageRank, the optimal strategy is unchanged from that of Figure 6.8.

Our next query result scenario and distortion measure are as follows. Let τ be the ordered list of the top 100 documents when query results are ranked by the product of the cosine similarity of the query to the document and the PageRank of the document: $cos_{\mathcal{Q}d} \cdot r_d$. Let τ_q be the ordered list of the top 100 documents when query results are ranked by $cos_{\mathcal{Q}d} \cdot q(r_d)$ for some quantizer q. Note that $\tau \neq \tau_q$ because $q(r_d)$ is less precise than r_d. We measure distortion using the KDist measure described in Section 6.2.2. As shown in Figure 6.11, in this scenario, the **log** strategy performs the best for all codelengths in minimizing the mean KDist distortion.

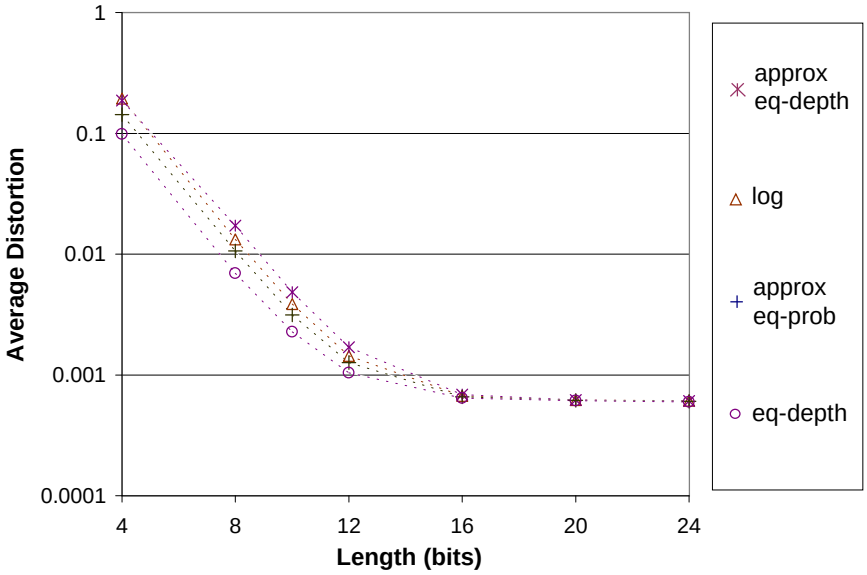


Figure 6.9: The approximate equiprobable partition outperforms the approximate equal-depth partition on the TDist distortion measure. Documents with high PageRank had slightly higher probabilities of being candidate results. The true equal-depth partition strategy still has the least distortion.

The previous results demonstrate the importance of using a distortion measure suited to the ranking function used by the search engine when choosing a quantization strategy.

6.5 Variable-Length Encoding Schemes

Fixed length encoding schemes are simple to support in the implementation of the ranking function, because the attribute values are at easily computed offsets into the attribute vector.

Variable-length encodings have the potential to reduce the *average* codeword lengths, and thus the overall storage requirement for the ranking vector. However, the downside is a more complex decoding process, which is less efficient and may not be practical, depending on the search engine's performance criteria. In particular,

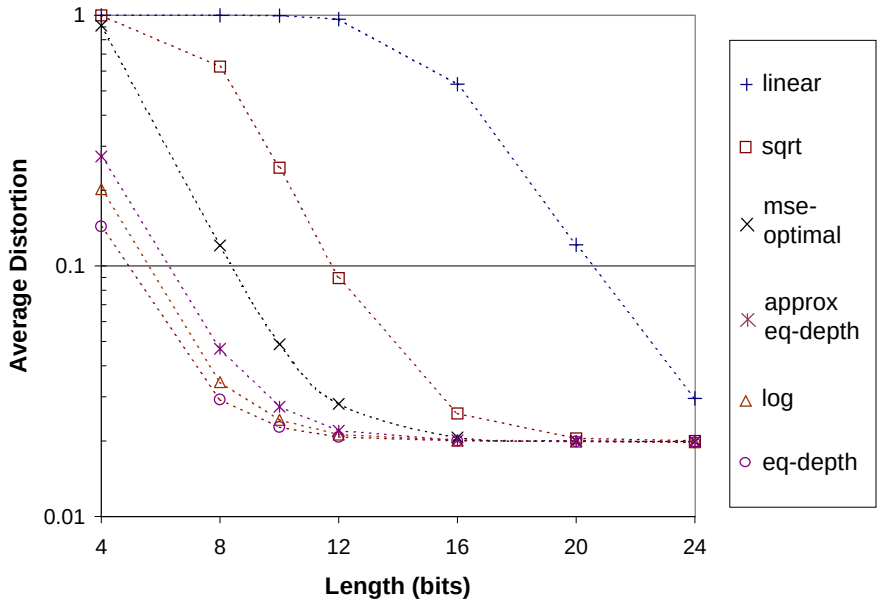


Figure 6.10: The average distortion when using the TDist distortion measure over pruned lists of query results, for 86 test queries. The pruned lists consisted of the top 100 results based on the cosine similarity of the documents to the query.

to retrieve the attribute values when the ranking vector is encoded with a variable-length scheme, a sparse index[16] is needed to allow the lookup of the block containing the desired value. Furthermore, all the values in that block preceding the desired value would also need to be decoded. In this section, we first explore the effectiveness of variable-length schemes in minimizing storage, and then investigate the additional runtime costs for decoding variable-length codes.

6.5.1 Variable-Length Encoding Performance

To investigate the effectiveness of variable-length schemes, we computed the average Huffman codelengths for the quantization schemes previously discussed in Sections 6.3 and 6.4. When the MSE distortion of Figure 6.5 is plotted against the Huffman codelength, rather than the fixed codelength, the uniform quantization strategy, **linear**,

[16]The index needs to be sparse, since otherwise any space savings from a variable-length coding scheme would be lost.

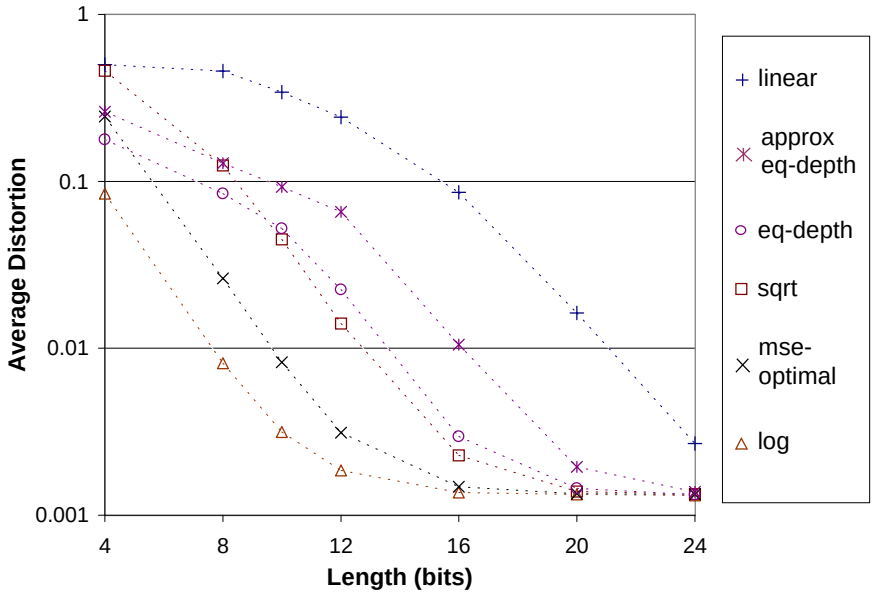


Figure 6.11: The average distortion when using the KDist distortion measure for 86 test queries. The rankings consisted of the top 100 results ranked by the score $cos_{Qd} \cdot r_d$.

becomes the best performer, as shown in Figure 6.12.[17] The variable-length encoding for the cells eliminates the inefficiencies of uniform quantization. This effect carries over to the result-based distortion measures as well, as shown in the replotting of Figure 6.8, given as Figure 6.13. Note that the equal-depth approaches derive no benefit from Huffman coding – the average codelength is reduced precisely when the cell depths are *non*uniform. Note that the average codeword lengths shown in these graphs does not include the memory required for the additional indexes needed at runtime for efficient decoding; we discuss decoding requirements in Section 6.5.2.

These empirical results indicate that if a variable-length encoding scheme is used to generate codes for the cells, a uniform quantizer performs similarly to the optimal quantizer, under the distortion measures we used.

[17]That uniform-width quantizers are near-optimal for variable-length encoding is known for the MSE distortion measure [28].

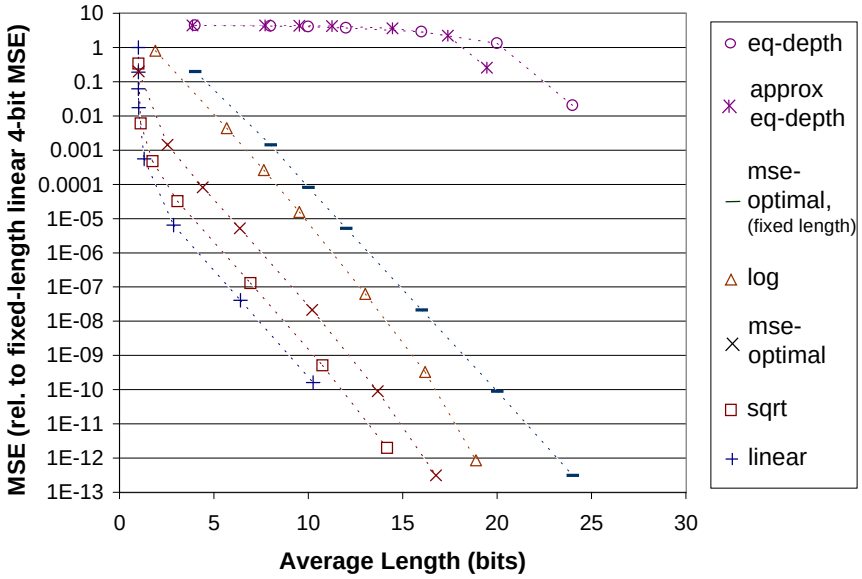


Figure 6.12: The MSE of 6 different strategies, plotted against the average codeword length used. The MSE axis is on a log scale, and is normalized so that the MSE of the 4-bit linear compander is 1. For comparison, the optimal fixed-length performer is also shown; the simple linear quantizer, using variable-length codes, performs better than the optimal fixed-length strategy.

6.5.2 Variable-Length Encoding Costs

Variable-length encoders outperform fixed-length encoders, when judged on the average codelength needed to achieve a particular distortion, for most of the distortion measures we have discussed. However, there is a processing overhead at query time to decode the numeric attribute values. The driving motivation behind our work was to reduce the per-document attribute lookup cost by fitting the ranking vectors in main memory; variable-length encodings are only appropriate if fixed-length encodings are not sufficient to allow the attribute vectors to be stored in memory. We next discuss the offline and query-time costs of variable-length schemes compared to fixed-length schemes.

During the offline step, compression of the input data values using variable-length schemes requires first generating Huffman codes for the cells of the partition, and then generating a compressed version of the input by replacing each input value with the Huffman codeword assigned to the cell the value was mapped to. A fixed-length

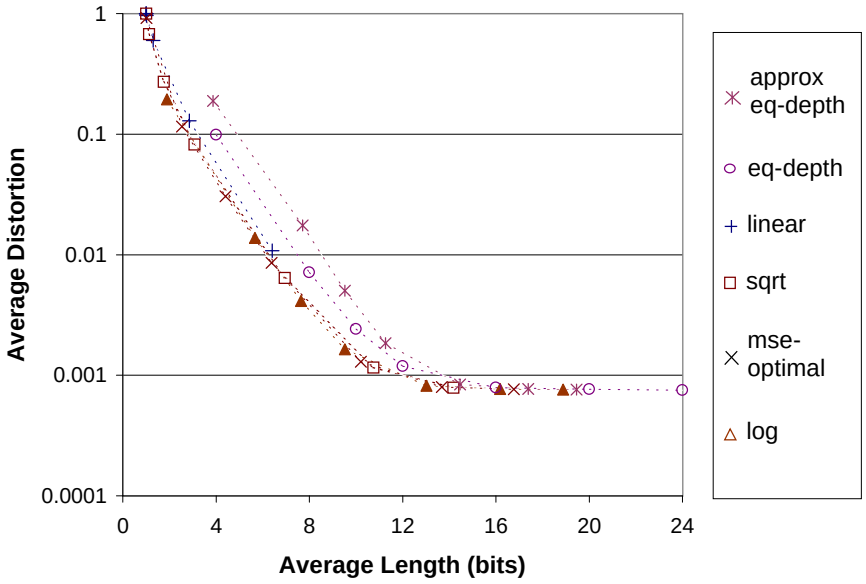


Figure 6.13: The average distortion for the variable length strategies when using the TDist distortion measure over the full lists of query results, for 86 test queries. The x-axis represents the *average* codeword lengths using a Huffman code.

scheme does not require generating a Huffman code — the intervals can be assigned sequential l-bit IDs. However, the cost of generating a Huffman code is fairly low; using the implementation of [57], we were able to generate the codebook and compress the input data (360M values, 1.34GB) in under 10 minutes using an AMD Athlon 1533MHz machine with a 6-way RAID-5 volume. Given the minimal impact of small variations in preprocessing cost, we did not explore further the offline overhead for variable-length encoding schemes.

The impact of additional *query-time* costs, however, is more significant. For both the fixed-length and variable-length scenarios, the query engine loads the entire sequence of quantized values into memory as a string b of bits. We assume that documents are identified by consecutively assigned document identifiers, so that the document with ID i is the ith value inserted into the bit string. In the case of a fixed-length scheme with codewords of length l, the attribute value associated with some document i is simply the value of the bit substring $b[i \times l, (i+1) \times l - 1]$. The only cost is a memory lookup; in the case where l is not the length of standard integer

data type (e.g., 8, 16, or 32), a few bit shifts are also required.

When using a variable-length code, however, for a given document i, finding the exact location for its attribute value in the bit string becomes nontrivial. Decoding from the beginning of the string until finding the ith value is of course inefficient, making an index necessary. More specifically, since maintaining the exact offset into the bitstring for each value would completely negate the benefit of compression, we must use a sparse index which maintains offsets to blocks of values. Decoding the attribute value for document i requires decoding all the values from the beginning of the block up through the desired value. Thus, the decoding time is proportional the block size B; more precisely, the expected number of decodes is $B/2$. Using small blocks reduces the decoding time, but in turn increases the space usage of the sparse index. Figure 6.14 shows the decode time, in μs/document vs. block size, for 4 variable-length schemes. For comparison, the decode time for a fixed-length encoding scheme is also given. These times were measured on an AMD Athlon 1533MHz machine with 2GB of main memory. The additional space overhead, in bits/codeword, needed by the sparse index for 360M values for various block sizes is plotted in Figure 6.15.

The times may seem very small, making the variable-length schemes seem attractive; however, for a large-scale search engine, with thousands of concurrent active queries, where each query has thousands of candidate results requiring attribute value decodings for tens of attributes, the per-result decode time needs to be as inexpensive as possible. As an illustrative example, consider a search engine with 1 billion pages with a query workload of 10 queries/s. Assume that each document has a single numeric property (e.g., PageRank) that needs to be decoded for calculating final rankings. Also assume that the average query yields .01% of the repository as candidate results, so that the processing for each query requires retrieving the numeric properties for 100,000 documents. If a variable-length scheme is used, so that the decode time for a single attribute value for a single document requires 35 μs, decoding alone will require 3.5 seconds of CPU time per query, or equivalently, 35 machines are needed to handle the query workload (if decoding were the only cost in the system). If the decode time is instead 1 μs per document (e.g., utilizing a fixed-length

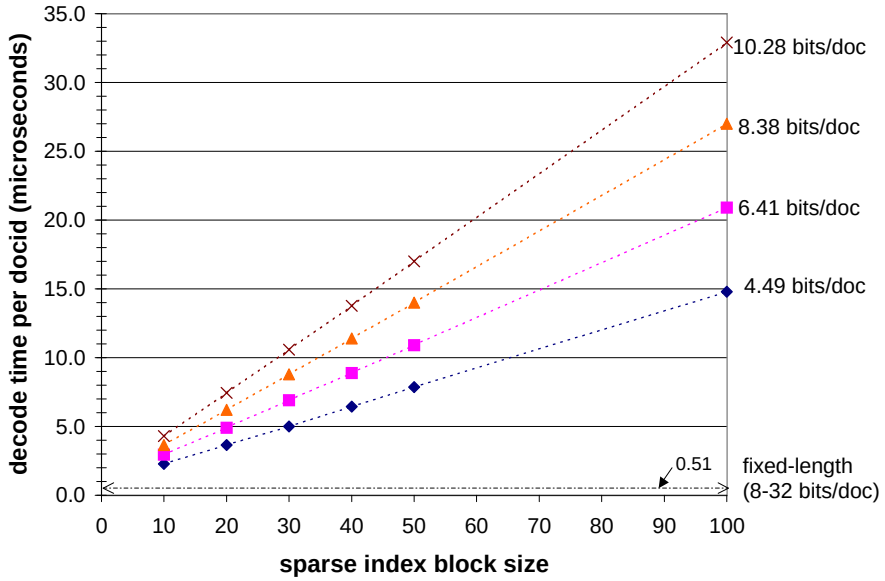


Figure 6.14: The decode time in microseconds per document for a PageRank vector quantized uniformly using variable-length codes with 4 different average codeword lengths. The decode time using a fixed-length code is also given for comparison.

encoding scheme), only 0.1s is spent decoding for each query; equivalently, a single machine can handle the query workload. Of course there are other significant costs in the system in addition to attribute value decode time; our goal in this example is simply to provide some intuition as to why per-document decode times need to be kept small.

6.6 Related Work

There has been much work in the field of compression in the context of large-scale Web search. An excellent overview of text-index compression techniques can be found in [74]. Suel and Yuan [72] investigate strategies for compressing the Web hyperlink graph. Raghavan and Garcia-Molina [67] explore techniques for compressing the Web link graph in ways that allow for efficient query processing.

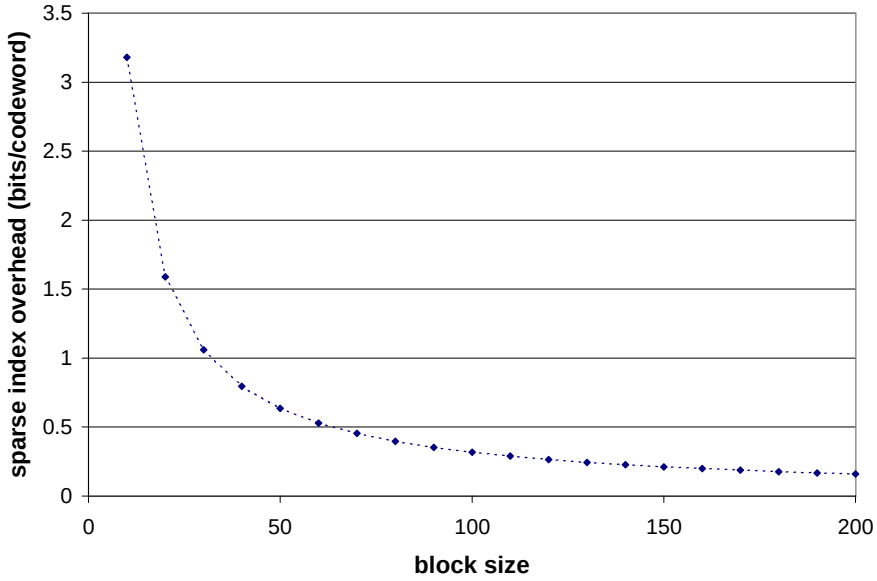


Figure 6.15: The additional space overhead needed by the sparse index, measured as bits/codeword, for block sizes ranging from 10 to 200.

Our work explores the development of lossy encodings for auxiliary numeric ranking vectors, where the quality of an encoding is judged by its effect on the final *rankings* induced over query results. An approach for efficiently encoding the document-length vector, needed for cosine computations, was studied in [58]. However, that work did not consider variable-length encodings, and did not provide analytic results for the behavior of the encodings under various models for query result distributions.

6.7 Conclusion

We have seen that simple fixed-length encoding schemes can substantially reduce the space needed for holding ranking indexes. For a search index of a billion pages, using a standard 4-byte single-precision floating point representation, each replica[18] of the index would require 4GB of main memory for *each* PageRank vector desired. A system utilizing 16 Topic-Sensitive PageRank values per page would thus require 64GB of

[18]For fast query response times, search engines generally utilize several many replicated search indexes.

main memory per replica. Using the **log** compander, which performs well with respect to all of the distortion criteria we considered, with 12-bit fixed length codewords, we can reduce the storage requirement to 24GB per replica. Further reductions in storage requirements, at the expense of additional complexity, are possible with the use of variable-length encoding schemes.

6.A Optimal Quantizer: Derivation Using the Multivariate Hypergeometric Distribution[19]

The derivation of the optimal quantizer under the distortion measure given in Equation 6.4, when the operation **Retrieve** is modeled as a uniform random sample, *without replacement*, is very similar to the derivation given in Section 6.4.2. As before, we want to minimize Equation 6.5, although each X_i now follows the hypergeometric, rather than the binomial, distribution. Using the known mean and variance for the hypergeometric distribution [29], with parameters N, N_i, and m, and letting $p_i = \frac{N_i}{N}$, we compute $E[X_i^2]$ as before:

$$E[X_i] = mp_i \tag{6.19}$$

$$var[X_i] \equiv E[X_i^2] - E[X_i]^2 \tag{6.20}$$

$$var[X_i] = mp_i(1-p_i)\frac{N-m}{N-1} \tag{6.21}$$

$$E[X_i^2] = mp_i(1-p_i)\frac{N-m}{N-1} + (mp_i)^2 \tag{6.22}$$

Plugging $E[X_i^2]$ back into Equation 6.5 and simplifying, we get

$$E[Distortion_m] = \frac{1}{m^2}\sum E[X_i^2] \tag{6.23}$$

$$= \frac{1}{m^2}\sum_i \big(mp_i(1-p_i)\frac{N-m}{N-1} + (mp_i)^2\big) \tag{6.24}$$

$$= \frac{1}{m^2}\sum_i \big(\frac{N-m}{N-1}(mp_i - mp_i^2) + m^2p_i^2\big) \tag{6.25}$$

$$= \frac{N-m}{m(N-1)}\sum_i p_i + \big(1 - \frac{N-m}{m(N-1)}\big)\sum_i p_i^2 \tag{6.26}$$

$$= \frac{N-m}{m(N-1)} + \big(1 - \frac{N-m}{m(N-1)}\big)\sum_i p_i^2 \tag{6.27}$$

Given values for N and m, the above is equivalent to minimizing $\sum_i p_i^2$ subject to the constraint $\sum_i p_i = 1$, leading to the solution given by Equation 6.15, i.e., an equal-depth partitioning scheme.

[19] Mayur Datar and Aristides Gionis provided assistance with the derivation that follows.

Bibliography

[1] 'More Evil Than Dr. Evil?'
http://searchenginewatch.com/sereport/99/11-google.html.

[2] E. Amitay. Using Common Hypertext Links to Identify the Best Phrasal Description of Target Web Documents. *Proceedings of SIGIR'98 Post-Conference Workshop on Hypertext Information Retrieval for the Web*, 1998.

[3] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin. PageRank computation and the structure of the web: Experiments and algorithms. In *Poster Proceedings of the Eleventh International World Wide Web Conference*, 2002.

[4] G. Attardi, A. Gull, and F. Sebastiani. Theseus: Categorization by context. *Proceedings of the Eighth World Wide Web Conference*, 1999.

[5] K. Bharat, B.-W. Chang, M. Henzinger, and M. Ruhl. Who links to whom: Mining linkage between web sites. In *Proceedings of the IEEE International Conference on Data Mining*, November 2001.

[6] K. Bharat and M. R. Henzinger. Improved algorithms for topic distillation in a hyperlinked environment. In *Proceedings of ACM SIGIR*, 1998.

[7] M. Bianchini, M. Gori, and F. Scarselli. Inside PageRank. *ACM Transactions on Internet Technology*, 2003.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[9] A. Broder. Filtering Near-duplicate Documents. *Proceedings of FUN*, 1998.

[10] A. Broder. On the Resemblance and Containment of Documents. *In Compression and Complexity of Sequences*, 1998.

[11] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise Independent Permutations. *Proceedings of STOC*, 1998.

[12] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic Clustering of the Web. *Proceedings of the Sixth International World Wide Web Conference*, 1997.

[13] S. Chakrabarti, B. Dom, D. Gibson, J. Kleinberg, P. Raghavan, and S. Rajagopalan. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[14] S. Chakrabarti, B. Dom, and P. Indyk. Enhanced Hypertext Categorization Using Hyperlinks. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998.

[15] S. Chakrabarti, M. M. Joshi, K. Punera, and D. M. Pennock. The structure of broad topics on the web. In *Proceedings of the Eleventh International World Wide Web Conference*, 2002.

[16] S. Chien, C. Dwork, R. Kumar, and D. Sivakumar. Towards exploiting link evolution. In *Proceedings of the Workshop on Algorithms for the Web*, November 2002.

[17] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proceedings of the Eleventh International World Wide Web Conference*, 2002.

[18] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding Interesting Associations without Support Pruning. *Proceedings of the International Conference on Data Engineering*, 2000.

[19] P.-J. Courtois. *Queueing and Computer System Applications*. Academic Press, 1977.

[20] B. Davison. Topical Locality in the Web. *Proceedings of ACM SIGIR*, 2000.

[21] J. Dean and M. Henzinger. Finding related pages in the world wide web. In *Proceedings of the Eighth International World Wide Web Conference*, 1999.

[22] M. Diligenti, M. Gori, and M. Maggini. Web page scoring systems for horizontal and vertical search. In *Proceedings of the Eleventh International World Wide Web Conference*, May 2002.

[23] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of the Tenth International World Wide Web Conference*, 2001.

[24] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[25] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1996.

[26] L. A. Goodman and W. H. Kruskal. Measures of association for cross classifications. *J. of Amer. Stat. Assoc.*, 49:732–764, 1954.

[27] Google. `http://www.google.com/`.

[28] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6), October 1998.

[29] G. Grimmett and D. Stirzaker. *Probability and Random Processes*. Oxford University Press, 1989.

[30] Z. Gyongyi, H. Garcia-Molina, and J. Pedersen. Combating web spam with TrustRank. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, 2004.

[31] T. Haveliwala, S. Kamvar, and G. Jeh. An analytical comparison of approaches to personalizing PageRank. *Stanford University Technical Report*, July 2003.

[32] T. Haveliwala, S. Kamvar, D. Klein, C. Manning, and G. Golub. Computing pagerank using power extrapolation. *Stanford University Technical Report*, July 2003.

[33] T. H. Haveliwala. Efficient computation of PageRank. *Stanford University Technical Report*, 1999.

[34] T. H. Haveliwala. Efficient encodings for document ranking vectors. *Stanford University Technical Report*, November 2002.

[35] T. H. Haveliwala. Topic-sensitive PageRank. In *Proceedings of the Eleventh International World Wide Web Conference*, May 2002.

[36] T. H. Haveliwala. Efficient encodings for document ranking vectors. In *Proceedings of the Fourth International Conference on Internet Computing*, June 2003.

[37] T. H. Haveliwala. Topic-sensitive PageRank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, pages 784–796, July/August 2003.

[38] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB)*, May 2000.

[39] T. H. Haveliwala, A. Gionis, D. Klein, and P. Indyk. Evaluating strategies for similarity search on the web. In *Proceedings of the Eleventh International World Wide Web Conference*, May 2002.

[40] T. H. Haveliwala and S. D. Kamvar. The second eigenvalue of the Google matrix. *Stanford University Technical Report*, 2003.

[41] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase: A repository of web pages. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[42] P. Indyk. A Small Minwise Independent Family of Hash Functions. *Proceedings of the Symposium on Discrete Algorithms*, 1999.

[43] M. Iosifescu. *Finite Markov Processes and Their Applications*. John Wiley and Sons, Inc., 1980.

[44] D. L. Isaacson and R. W. Madsen. *Markov Chains: Theory and Applications*, chapter IV, pages 126–127. John Wiley and Sons, Inc., New York, 1976.

[45] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

[46] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Exploiting the block structure of the web for computing PageRank. *Stanford University Technical Report*, 1999.

[47] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating PageRank computations. In *Proceedings of the Twelfth International World Wide Web Conference*, May 2003.

[48] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[49] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, September 1999.

[50] A. N. Langville and C. D. Meyer. Updating PageRank with iterative aggregation. In *Proceedings of the Thirteenth International World Wide Web Conference, Poster Track*, 2004.

[51] A. N. Langville and C. D. Meyer. Deeper inside PageRank. *Internet Mathematics*, 5, 2005.

[52] C. P.-C. Lee, G. H. Golub, S. A. Zenios, and S. Leuyng. A fast two-stage algorithm for computing PageRank. *Stanford University Technical Report*, 2003.

[53] H. P. Luhn. The Automatic Creation of Literature Abstracts. *IBM Journal of Research and Development*, 2:159–165, 1958.

[54] D. McAllister, G. Stewart, and W. Stewart. On a Rayleigh-Ritz refinement technique for nearly uncoupled stochastic matrices. *Linear Algebra and Its Applications*, 60:1–25, 1984.

[55] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proceedings of the Tenth International World Wide Web Conference*, 2001.

[56] T. Mitchell. *Machine Learning*, chapter 6, pages 177–184. McGraw-Hill, Boston, 1997.

[57] A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In S. Akl, F. Dehne, and J.-R. Sack, editors, *Proceedings of the Workshop on Algorithms and Data Structures*, pages 393–402, Queen's University, Kingston, Ontario, Aug. 1995. LNCS 955, Springer-Verlag.

[58] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Information Processing and Management*, 30(6):733–744, November 1994.

[59] National Institute of Standards and Technology (NIST). *The Eighth Text Retrieval Conference (TREC-8)*, 1999.

[60] Open Directory Project (ODP). `http://www.dmoz.com/`.

[61] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. *Stanford Digital Libraries Working Paper*, 1998.

[62] G. Pandurangan, P. Raghavan, and E. Upfal. Using PageRank to characterize web structure. In *Proceedings of the Eighth Annual International Computing and Combinatorics Conference*, 2002.

[63] P. Panter and W. Dite. Quantization in pulse-count modulation with nonuniform spacing of levels. In *Proceedings of IRE*, Jan. 1951.

[64] D. Pennock, G. Flake, S. Lawrence, E. Glover, and C. L. Giles. Winner's don't take all: Characterizing the competition for links on the web. In *Proceedings of the National Academy of Sciences*, 2002.

[65] M. Porter. An Algorithm for Suffix Stripping. *Program: Automated Library and Information Systems*, 14(3):130–137, 1980.

[66] D. Rafiei and A. O. Mendelzon. What is this page known for? Computing web page reputations. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[67] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *Proceedings of the IEEE International Conference on Data Engineering*, March 2003.

[68] M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in PageRank. In *Advances in Neural Information Processing Systems*, volume 14. MIT Press, Cambridge, MA, 2002.

[69] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[70] S. Siegel and N. J. Castellan. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill, 1988.

[71] H. A. Simon and A. Ando. Aggregation of variables in dynamic systems. *Econometrica*, 29:111–138, 1961.

[72] T. Suel and J. Yuan. Compressing the graph structure of the web. In *IEEE Data Compression Conference (DCC)*, March 2001.

[73] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, 1965.

[74] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, San Francisco, 1999.

[75] Yahoo! `http://www.yahoo.com/`.

[76] H. Zhang, A. Goel, R. Govindan, K. Mason, and B. V. Roy. Making eigenvector-based repuation systems robust to collusions. In *Proceedings of the Workshop on Algorithms and Models for the Web Graph*, 2004.