RankMass Crawler: A Crawler with High Personalized PageRank Coverage Guarantee

Junghoo Cho
University of California Los Angeles
4732 Boelter Hall
Los Angeles, CA 90095
cho@cs.ucla.edu

Uri Schonfeld
University of California Los Angeles
4732 Boelter Hall
Los Angeles, CA 90095
shuri@shuri.org

ABSTRACT

Crawling algorithms have been the subject of extensive research and optimizations, but some important questions remain open. In particular, given the unbounded number of pages available on the Web, search-engine operators constantly struggle with the following vexing questions: *When can I stop downloading the Web? How many pages should I download to cover "most" of the Web? How can I know I am not missing an important part when I stop?* In this paper we provide an answer to these questions by developing, in the context of a system that is given a set of trusted pages, a family of crawling algorithms that (1) provide a theoretical guarantee on how much of the "important" part of the Web it will download after crawling a certain number of pages and (2) give a high priority to important pages during a crawl, so that the search engine can index the most important part of the Web first. We prove the correctness of our algorithms by theoretical analysis and evaluate their performance experimentally based on 141 million URLs obtained from the Web. Our experiments demonstrate that even our simple algorithm is effective in downloading important pages early on and provides high "coverage" of the Web with a relatively small number of pages.

1. INTRODUCTION

Search engines are often compared based on how much of the Web they index. For example, Google claims that they currently index 8 billion Web pages, while Yahoo states its index covers 20 billion pages [5]. Unfortunately, comparing search engines based on the sheer number of indexed pages is often misleading because of the unbounded number of pages available on the Web. For example, consider a calendar page that is generated by a dynamic Web site. Since such a page often has a link to the "next-day" or "next-month" page, a Web crawler[1] can potentially download an unbounded number of pages from this single site by following an unbounded sequence of next-day links. Thus, 10 billion pages indexed by one search engine may not have as many "interesting" pages as 1 billion pages of another search engine if most of its pages came from a single dynamic Web site.

[1]A crawler is an automatic program that follows links and downloads pages for a search engine.

The intractably large size of the Web introduces many important challenges. In particular, it makes it *impossible* for any search engine to download the entire Web; they *have to stop* at some point. But exactly when should they stop? Are one billion pages enough or should they download at least 10 billion pages? How much of the "interesting" content on the Web do they miss if they index one billion pages only? In this paper, we formally study the following two issues related to these questions:

1. *Crawler coverage guarantee*: Given the intractably large size of the Web search-engine operators wish to have a certain guarantee on how much of the "important" part of the Web they "cover" when they stop crawling. Unfortunately, providing this guarantee is challenging because it requires us to know the importance of the pages that have *not* been downloaded. Without actually "seeing" these pages, how can a crawler know how important the pages are?

2. *Crawler efficiency*: A crawler also wishes to download "important" pages early during a crawl, so that it can cover the most important portions of the Web when it stops. How can a crawler achieve this goal and attain a certain coverage with the minimum number of downloads? Alternatively, how can it achieve the maximum coverage with the same number of downloads?

In the rest of this paper, we investigate these issues in the following order:

- In Section 2, we first formalize the notion of the coverage of a set of Web pages using the metric called *RankMass*. Our RankMass metric is similar to the one proposed in [19] to compare the quality of search-engine indexes. RankMass is based on a commonly-used variation of PageRank metric (often referred to as *Personalized PageRank* or *TrustRank*) which assumes that the users' random jumps are limited only to a set of trusted pages. We then provide a formal and reasonable assessment on how much of the "interesting" content on the Web the set covers.
- In Section 3, we prove key theorems that provide a formal way to measure the RankMass coverage of a particular

subset of the Web. This theoretical result is particularly useful because the coverage measurement can be done just based on the pages within the subset, *without* knowing anything about the pages outside of the subset. We then describe a simple yet effective crawling algorithm that provides a tight coverage guarantee of the downloaded pages as long as the seed set of the crawl captures the majority of the pages that users randomly jump to.

- In Section 4, we then describe how the crawler can prioritize the pages during crawling, so that it can download high Personalized PageRank pages early during a crawl, and achieve the highest RankMass when it stops. Again, the crawling algorithm is developed through a careful analysis of the properties of the Personalized PageRank metric and is able to provide a strict guarantee on the downloaded RankMass.
- In Section 5, we evaluate the effectiveness of our crawling algorithms by running extensive experiments on real Web data. Our results indicate that even our simple algorithm performs well in practice and is able to achieve high RankMass coverage early during a crawl.

There exist a large body of related work in the literature that investigates related issues [25, 3, 15, 11, 14] and in particular instances of the family of algorithms we present in this paper have been described before [24, 2, 12]. However, due to the fundamental difficulty of computing exact Personalized PageRank values without downloading the entire Web, previous studies are based on a set of intuitive heuristics and/or analysis that is mainly focused on the eventual convergence of the algorithms. As far as we know our work is the first that provides a strict Personalized PageRank coverage guarantee on the downloaded pages during and after the crawl. This allows search-engine operators to judge exactly how much of the Web they have covered so far, and thus make an informed decision on when it is safe to stop crawling the Web. In the next section we start by introducing the RankMass coverage metric.

2. RANKMASS COVERAGE METRIC

The Web can be viewed as a potentially infinite set of documents, denoted by D. The Web crawler can only download a finite number of documents D_C from D. In this section, we discuss how we may define the measure of the coverage achieved by D_C on a potentially infinite set D.

We represent the importance of each page by assigning the weight w_i to $p_i \in D$, where w_i values are normalized to be $\sum_{p_i \in D} w_i = 1$. Then the coverage of D_C can be defined as its *mass*, the sum of the weights of its pages: $\sum_{p_i \in D_C} w_i$. For example, assigning every page p_i with a weight of $\frac{1}{2^i}$ will result in the whole Web's mass being $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$, and the mass of the first 10 pages to be $\sum_{i=1}^{10} \frac{1}{2^i} = 1 - \frac{1}{2^{10}}$

The important question is exactly how we should assign weights to Web pages? What will be the fair and reasonable weight for each page? Many possible weight assignments may be considered. For example, a document collection may be evaluated according to its relevance to a particular query or set of queries. Although this metric may be interesting, it would require us to know these set of queries beforehand in order to evaluate a document collection. Other weight assignments might leverage the page topic, page size, the fonts used or any other features associated with the document.

In this paper we choose to explore one such weight assignment, where each page is associated with a weight equal to its PageRank, similarly to the work done by Heydon et al. [19]. There are a number of reasons why PageRank is useful as a measure of page importance. Since its first introduction in 1996, PageRank has proven to be very effective in ranking Web pages, so search engines tend to return high PageRank pages higher than others in search results. It is also known that under the random-surfer model, the PageRank of a page represents the probability that a random Web surfer is at the page at a given time, so high PageRank pages are the ones that the users are likely to look at when they surf the Web. We now briefly go over the definition of PageRank and the random-surfer model to formally define our coverage metric.

2.1 Review of the PageRank metric

Original PageRank Let D be the set of of all Web pages. Let $\mathcal{I}(p)$ be the set of pages that link to the page p and let c_i be the total number of links going out of page p_i. The PageRank of page p_i, denoted by r_i, is then given by[2]

$$r_i = d \left[\sum_{p_j \in \mathcal{I}(p_i)} \frac{r_j}{c_j} \right] + (1-d)\frac{1}{|D|} \tag{1}$$

Here, the constant d is called a *damping factor* which is often assumed to be 0.85. Ignoring the damping factor for now, we can see that r_i is roughly the sum of r_j's that point to p_i. Under this formulation, we construct one equation per Web page p_i with the equal number of unknown r_i values. Thus, the equations can be solved for the r_i values, assuming that the PageRank values sum up to one: $\sum_{i=1}^{n} r_i = 1$.

A way to think intuitively about PageRank is to consider a user "surfing" the Web, starting from any page, and randomly selecting a link from that page to follow. When the user is on page p_j, with the probability d, the user will click on one of the c_j links on the page with equal probability, meaning $\frac{1}{c_j}$. With the remaining probability, $1-d$, the user stops following links, and jumps to a random page in D. This damping factor d makes sense because users will only continue clicking on links for a finite amount of time before they get distracted and start exploring something completely unrelated.

A more compact notation can be achieved by using matrices as follows:

$$\vec{R} = d \left[\mathbf{W} \cdot \vec{R} \right] + (1-d)\frac{1}{|D|}\vec{1} \tag{2}$$

In this equation $\mathbf{W}$ is the stochastic transition matrix whose (i,j) element w_{ij} is $w_{ij} = \frac{1}{c_j}$, if a link exists from page p_j to page p_i and $w_{ij} = 0$ otherwise. $\vec{R}$ is a vector whose ith element is r_i, the PageRank of p_i. Finally, $\vec{1}$ is a vector whose elements are all 1.

Personalized PageRank The original PageRank formulation suffers from easy spammability because a malicious Web publisher can significantly boost the PageRank of his pages simply by generating a massive number of pages and

[2]For clarify, here we assume that D is a finite set. Under the personalized PageRank definition that we describe later, our results can be easily generalized to the case when D is infinite.

heavily linking among themselves. To avoid the spammability problem, a variation of the original PageRank has been proposed [20, 16, 17] and is being commonly used.

Under this model, every page p_i is associated with a *trust score* t_i (here, t_i values are assumed to be normalized to $\sum_{p_i \in D} t_i = 1$). We refer to the vector of the t_i values as the *trust vector* $\vec{T}$:

$$\vec{T} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \end{bmatrix} \tag{3}$$

The PageRank of p_i is then defined as

$$r_i = d \left[\sum_{p_j \in \mathcal{I}(p_i)} \frac{r_j}{c_j} \right] + (1-d)t_i, \tag{4}$$

or using the matrix notation,

$$\vec{R} = d \left[\mathbf{W} \cdot \vec{R} \right] + (1-d)\vec{T}. \tag{5}$$

Note that the only difference between Equations 1 and 4 is that the factor $1/|D|$ in the first equation is replaced with t_i in the second one. Thus, if $t_i = 1/|D|$ for every p_i, the personalized PageRank reduces to the original one. In general, t_i is set to a non-zero value only for the page p_i that is "trusted" or believed to be "non-spam" by the search engine. Intuitively, this personalized PageRank incorporates the fact that when a user gets "interrupted" and "jumps" to a new page, he does not jump to every page with equal probability. Rather, the user is more likely to jump to the pages that he "trusts"; the trust score t_i represents this random jump probability to p_i.

It is well known that when t_i is set to non zero only for non-spam pages, a spammer cannot increase r_i by simply creating a large number of pages that link to p_i. Thus, it is significantly more difficult to spam the personalized PageRank when t_i values are carefully chosen. In fact, even the original Google prototype is known to have used the personalized PageRank, giving higher t_i values to the pages in the educational domain. In the rest of this paper we assume the personalized PageRank and simply refer to it as PageRank for brevity.

2.2 RankMass coverage metric

We now formalize our coverage metric:

Definition 1 (RankMass) Assume D is the set of all pages on the Web and $\vec{T}$ is the trust vector. For each $p_i \in D$, suppose r_i represents the PageRank of p_i computed based on the *entire* link structure within D using $\vec{T}$. Then the *RankMass* of a subset $D_C \subset D$, $RM(D_C)$, is defined as the sum of its PageRank values:

$$RM(D_C) = \sum_{p_i \in D_C} r_i \qquad \square$$

Given the random-surfer interpretation of PageRank, $RM(D_C)$ represents the probability of random surfers' visiting pages in D_C. For example, if $RM(D_C)$ is 0.99, then 99% of the users are visiting the pages in D_C at any particular time. Therefore, by guaranteeing a high RankMass for D_C, we guarantee that most of the pages that users will surf to are in D_C. For the reader's convenience, we summarize all the symbols used in this paper in Table 1.

Symbol	Description
D	All documents on the Web
D_C	The set of documents obtained by crawling
d	Damping factor used in PageRank calculation
$\mathcal{I}(p)$	Set of pages that point to page p
c_i	Number of outgoing links from p_i
r_i	The PageRank of page p_i
$\vec{R}$	Vector of PageRank values
$\vec{T}$	A trust vector where t_i signifies the degree of trust we give to page p_i
$\mathbf{W}$	The transition matrix where w_{ij} signifies the probability that a random surfer will follow the link in page p_j to page p_i.
$RM(G)$	RankMass measure defined over a set of pages G as the sum of their PageRank values
$\mathcal{N}_L(G)$	The set of pages that are at most L links away from some page $p \in G$
$PP(w)$	Path probability of path w

Table 1: Summary of notation used

2.3 Problem statement

We now formally state the two problems that we investigate in this paper.

Problem 1 (Crawling algorithm with RankMass guarantee) Design an effective crawling algorithm that guarantees that when we stop downloading the Web, the RankMass of the downloaded pages D_C is close to one. That is, given ϵ, we want to develop a crawling algorithm that downloads a D_C such that

$$RM(D_C) = \sum_{p_i \in D_C} r_i \geq 1 - \epsilon \qquad \square$$

Problem 2 (Efficient crawling algorithm) Given the target download size $N = |D_C|$, design a crawling algorithm that maximizes the RankMass of the downloaded pages D_C. That is, given N, we want to develop a crawling algorithm that downloads pages such that $RM(D_C)$ is maximum out of all possible D_C's with $|D_C| = N$. $\square$

In addressing these problems, it is important to note that RankMass is computed based on the the link structure of the *entire* Web D, not just the graph structure within the subset D_C. Therefore, without downloading the entire Web, it is not possible to compute $RM(D_C)$ precisely. Then how can we design a crawling algorithm with the above properties? We address this issue in the next few sections.

3. CRAWLING WITH RANKMASS GUARANTEE

The primary goal of this section is to design a crawling algorithm that provides the RankMass coverage guarantee when it stops (Problem 1). As we briefly discussed before, the core challenge in addressing this problem is that the crawler does not know the exact RankMass of the downloaded pages D_C just based on the link structure in D_C. Then how can it provide the RankMass guarantee of D_C?

Our main idea for addressing this challenge in this paper is as follows: while the exact PageRank is difficult to compute, *we can still compute the lower bound of the r_i values for $p_i \in D_C$ just based on the link structure within D_C.* Based on this general idea, in Section 3.1 we first derive the lower bound of the download pages D_C for a special case where only a single page is trusted. Then in Section 3.2 we build on this result to compute the lower bound for the general case where multiple pages are trusted. Based on this result, we then describe a crawling algorithm that provides the RankMass guarantee. For the clarify of discussion, we assume that there exist no dangling page on the Web in the rest of this paper. It is relatively straightforward to extend our result to the Web with dangling pages, because the generalization can be done by additional step of renormalization [7].

3.1 RankMass lower bound under single-page trust vector $\vec{T}^{(1)}$

In this section, as a first step of designing a crawling algorithm with the RankMass guarantee, we investigate the RankMass lower bound of a set of pages for a very special case: here, we assume that the PageRank values are computed under a special trust vector $\vec{T}^{(1)}$ whose ith element, $t_i^{(1)}$, is defined as follows:

$$t_i^{(1)} = \begin{cases} 1 & \text{for } i = 1 \\ 0 & \text{for } i = 2, 3, \ldots \end{cases}$$

Note that the meaning of this trust vector, under the random-surfer interpretation, corresponds to the case when the user trusts only page p_1, and jumps only to this page when he gets "interrupted". While the RankMass lower bound computed for this special trust vector may not be directly useful for the general case when multiple pages are trusted, it provides the key insight for computing the lower bound for the general case.

As we will see, our RankMass lower bound is given for a particular set of pages that are in the "neighborhood" of p_1. Thus, we first formalize the notion of neighborhood:

Definition 2 We define the *L-neighbors* of page p, $\mathcal{N}_L(p)$, as the set of pages that are reachable from p by following L or fewer links. Similarly, given a group of pages G, the L-neighbors of the group G is defined as $\mathcal{N}_L(G) = \bigcup_{p \in G} \mathcal{N}_L(p)$. That is, it includes all pages reachable from any of the pages in G in L links or less. □

Given the above definition, we are now ready to state this section's main theorem:

Theorem 1 *Assuming the trust vector $\vec{T}^{(1)}$, the sum of the PageRank values of all L-neighbors of p_1 is at least d^{L+1} close to 1.*

$$\sum_{p_i \in \mathcal{N}_L(p_1)} r_i \geq 1 - d^{L+1} \tag{6}$$

□

Due to space limitations, we omit the full proof of this theorem in this version of the paper and just provide the intuition behind this theorem using the random-surfer model.

The theorem states that the sum of the PageRank values of the pages in $\mathcal{N}_L(p_1)$ is above a certain value. In other words, the probability of finding a random surfer in the L-neighborhood of p_1 is greater than some value. To help our explanation, in Figure 1 we show L-neighborhoods of p_1 for $L = 1, 2, 3$. Since our random surfer keeps returning to p_1 the chances of being in any page is the chance of being in p_1 times the chances of following all possible paths to that page. Furthermore, the longer the path from p_1 is, the less likely it is to be followed since our random surfer has many chances of being "interrupted". In particular, the probability that the surfer ventures out to *any* page not within L links from p_1 is at most d^{L+1} because the user should make $L + 1$ or more consecutive clicks, which is the key reason for the term d^{L+1} in Equation 6.

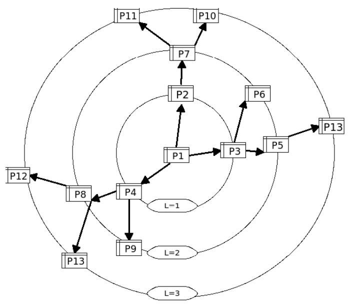


Figure 1: Neighborhoods of a single page

We now illustrate the implication of Theorem 1. According to the theorem, the RankMass of, say, $\mathcal{N}_3(p_1)$ is *at least* $1 - d^4$. Note that the theorem does not assume anything about the link structure of the pages. That is, this lower bound is *always* guaranteed under the trust vector $\vec{T}^{(1)}$, regardless of the link structure of the Web. Therefore, the theorem suggests an effective crawling strategy that guarantees the RankMass of the downloaded pages: starting from p_1, we simply download all pages reachable within L links from p_1! The theorem then guarantees that the pages in the downloaded collection have at least $1 - d^{L+1}$ of the RankMass of the entire Web. Unfortunately, search engines are unlikely to use only a single trusted page in their PageRank computation — for example, in [16], Gyongyi et al. reports using close to 160 pages as the trusted pages — so this algorithm is unlikely to be useful in practice. The next section extends the result of this section to the general cases when multiple pages are trusted.

3.2 RankMass lower bound under multi-page trust vector

We now explain how we can compute the RankMass lower bound of downloaded pages when the trust is spread across *a (possibly infinite) set of pages.* That is, we compute the RankMass lower bound for a general trust vector $\vec{T}$, where multiple t_i's may take non-zero values.

In order to state the RankMass lower bound for this general case, let G be the set of trusted pages (i.e., $G = \{p_i | t_i > 0\}$). Then the following theorem shows the RankMass lower bound:

Theorem 2 *The RankMass of the L-neighbors of the group of all trusted pages G, $\mathcal{N}_L(G)$, is at least d^{L+1} close to 1. That is,*

$$\sum_{p_i \in \mathcal{N}_L(G)} r_i \geq 1 - d^{L+1}. \tag{7}$$

□

Proof We prove this is by "decomposing" each page's PageRank value to individual values attributed to individual trusted pages. More precisely, assume $\vec{T}^{(k)}$ is the trust vector whose

kth element is the only non-zero element:

$$t_i^{(k)} = \begin{cases} 1 & \text{for } i = k \\ 0 & \text{for all other } i\text{'s.} \end{cases}$$

Also, let $\vec{R}^{(k)}$ be the PageRank vector computed under the trust vector $\vec{T}^{(k)}$. Then from the linearity property of PageRank, it can be shown that [18]:

$$\vec{R} = \sum_{p_i \in G} t_i \, \vec{R}^{(i)} \tag{8}$$

or

$$r_j = \sum_{p_i \in G} t_i \, r_j^{(i)} \quad \text{for any } p_j. \tag{9}$$

If we take the sum of the above equation for every $p_j \in \mathcal{N}_L(G)$, we get

$$\sum_{p_j \in \mathcal{N}_L(G)} r_j = \sum_{p_i \in G} t_i \sum_{p_j \in \mathcal{N}_L(G)} r_j^{(i)} \tag{10}$$

From Theorem 1, we know that $\sum_{p_j \in \mathcal{N}_L(G)} r_j^{(i)} \geq 1 - d^{L+1}$ for any i. We also know that $\sum_{p_i \in G} t_i = 1$ because G is the set of *all* trusted pages. Therefore, the above equation becomes

$$\sum_{p_j \in \mathcal{N}_L(G)} r_j \geq 1 \cdot (1 - d^{L+1}) = 1 - d^{L+1}. \tag{11}$$

■

We are now ready to introduce our crawling algorithm that provides a RankMass coverage guarantee for a general trust vector $\vec{T}$.

3.3 L-neighbor Crawler

The result of Theorem 2 suggests a surprisingly simple crawling algorithm that can provide a formal RankMass coverage guarantee: *download all pages within the L-neighbor of the set of trusted pages G!*

In Figure 2 we describe our L-neighbor algorithm that implements this idea. The algorithm initializes the first level of the crawl with the trusted pages in line 2. It then proceeds to crawl all this level's immediate neighbors at line 4. This process continues until the stopping condition $\epsilon < d^{L+1}$ is met at line 3.

```
1: $L := 0$
2: $N[0] = \{p_i | t_i > 0\}$ // Start with the trusted pages
3: While ($\epsilon < d^{L+1}$)
4:     Download all uncrawled pages in $N[L]$
5:     $N[L+1]$ = {all pages linked to by a page in $N[L]$}
6:     $L = L + 1$
```

Figure 2: L-neighbor Crawling Algorithm.

Given Theorem 2, it is easy to see that the L-neighbor algorithm provides the following guarantee when it stops.

Corollary 1 *When the L-neighbor algorithm stops, the RankMass of the downloaded pages is at least ϵ close to 1. That is, $RM(D_C) \geq 1 - \epsilon$.* □

Note that the L-neighbor algorithm is essentially the same as the well known BFS algorithm except that (1) our crawl is seeded with the the trusted set of nodes, (2) we have a clear, goal driven stopping condition and (3) we can give a formal bound on the "importance" of the (potentially infinite) set of undownloaded pages.

Finally, we discuss one potential problem when a search engine operator's trusted set of pages G is infinite — for example, given the random-surfer interpretation of t_i, a search engine operator may decide to analyze users' real Web-trace data to estimate their random jump probability to p_i, and use this estimated probability as the t_i. In this case, t_i can be non-zero for any p_i, at least in principle.

When the set of trusted pages G is infinite, its L-neighbor $\mathcal{N}_L(G)$ also contains an infinite number of pages, so the L-neighbor crawler cannot download all of these pages; it can only download the L-neighbor of a *finite subset* $G' \subset G$. Fortunately, even in this case, we can show that the algorithm still provides the following RankMass guarantee:

Theorem 3 *Assume that we have a finite set $G' \subset G$, whose trust score sum is $\sum_{p_i \in G'} t_i = 1 - \delta$. Then the RankMass of the L-neighbor of G' is*

$$\sum_{p_i \in \mathcal{N}_L(G')} r_i \geq (1 - \delta)(1 - d^{L+1}) \tag{12}$$

□

The proof can be done using the same steps of the proof for Theorem 2. The implication of this theorem is that in case a crawl can be seeded with a finite subset of trusted pages, the crawler's RankMass coverage depends both on the depth of the crawl and the sum of the trust scores of the seed pages. Later in our experiment section, we investigate the performance of the L-neighbor algorithm when the crawler is seeded with only a subset of trusted pages.

4. CRAWLING TO ACHIEVE HIGH RANKMASS

So far we have focused on our first problem, how to provide a coverage guarantee when the crawler stops. We now turn our attention to our second problem (Problem 2 in Section 2.3): *How can we download high PageRank pages early during a crawl, so that we can maximize the RankMass of the downloaded pages when we stop?*

The L-neighbor algorithm is simple and allows for an efficient implementation, but it is not directly suitable for the purpose of downloading high PageRank pages early. For example, consider running the L-neighbor algorithm with two seed pages, p_1 and p_2. p_1 has the trust value of 0.99, and p_2 has the trust value of 0.01. Intuitively, we expect the pages near p_1 to have much higher PageRank values than the ones near p_2. Given this intuition, we may want to modify the L-neighbor algorithm to download a larger neighborhood of p_1 earlier than that of p_2, so that the higher PageRank pages are downloaded sooner. As this example demonstrates, what our L-neighbor algorithm needs is more fine-grained page prioritization mechanism for use during the crawl.

In general, the core challenge of developing such a page prioritization mechanism is estimating the PageRank of the undownloaded pages during a crawl, especially given that we have only a partial subgraph of the Web. We explain our idea for addressing this challenge in the next section.

4.1 PageRank lower bound of an individual page

Computing the exact PageRank of an undownloaded page just from the downloaded part of the Web is not possible. Therefore, in the algorithms we present in the next few sections, we instead compute a *PageRank lower bound* for each page, and then, *give high priority for download to the pages*

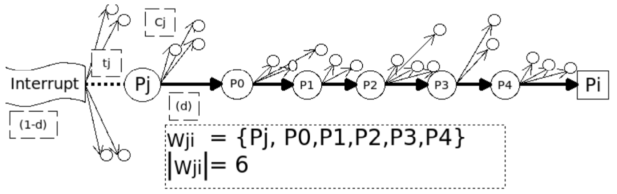


Figure 3: Example surfing session from P_j to P_i

with high PageRank lower bounds. The intuition behind this prioritization is that the pages with high lower bounds are more likely to have high PageRank values, although there is no strict guarantee that this will always be the case.

In this section, we explain how the PageRank lower bound can be computed just from a downloaded subset of the Web. However, before we provide our formal results, we first intuitively explain how this can be done based on the random surfer model introduced earlier.

As discussed earlier, the PageRank of a page p_i can be viewed as the probability that a random surfer is found at any given moment viewing this page. The random surfer follows a very simple path; with probability d he continues his current *surfing session*, *uninterrupted*, clicking on any link from the current page with equal probability. However, with probability $(1-d)$, the random surfer is *interrupted* and the current surfing session ends. In this case, the random surfer chooses a new random page p_j, with a probability equal to his trust in the page, t_j, and he begins a new surfing session starting from the page.

Given this model, how can we calculate the probability of the random surfer being in page p_i? In order to calculate this probability, we consider a user who is currently at p_i, and examine all the possible ways any session could have led him there. We show one possible such a session in Figure 1, where the user jumps to p_j in the beginning of the session and eventually comes to p_i. What is the probability of following such a path? To compute this probability, let w_{ji} denote *the sequence* of pages along it, *not including* the last page p_i itself. Thus, $|w_{ij}| = \sum_{p \in w_{ij}} 1$ is equal to the number of links the random surfer clicks in this session. At the beginning of this path, our random surfer was interrupted (with probability $1-d$), randomly jumped to page p_j (with probability t_j), and continued clicking on the right links along the way (with probability $\prod_{p_k \in w_{ji}} \frac{1}{c_k}$) without being interrupted throughout the session (with probability $d^{|w_{ji}|}$). Factoring in all of these together gives us:

$$\left(\prod_{p_k \in w_{ji}} \frac{1}{c_k} \right) (1-d) t_j d^{|w_{ji}|} \qquad (13)$$

This is the probability of one specific path w_{ji} corresponding to one specific surfing session that leads to p_i. We refer to this probability as the *path probability* of w_{ji}, which is denoted by $PP(w_{ji})$. Then, by summing up the path probabilities of all possible surfing sessions leading to p_i we get the probability of the random surfer being in p_i, which is PageRank. The following lemma formalizes this notion:

Lemma 1 *Let D be the set of all the pages on the Web, and W_{ji} be the set of all paths on the web leading from page p_j to page p_i. Thus the following is an alternative method of calculating the PageRank value of p_i denoted by r_i:*

$$r_i = \sum_{p_j \in D} \sum_{w_{ji} \in W_{ji}} PP(w_{ji}) \qquad (14)$$

$$PP(w_{ji}) = \left(\prod_{p_k \in w_{ji}} \frac{1}{c_k} \right) (1-d) t_j d^{|w_{ji}|} \qquad (15)$$

where c_k is the number of outgoing links from page p_k, t_j is the trust value for page p_j, and d is the damping factor. □

The detailed proof of this lemma is not included in this version of the paper due to space limitations. This lemma has also been independently proven by Brinkmeir in [9].

Let us see how this lemma can be used to calculate the lower bound for PageRank values. The lemma states that if we sum all these path probabilities $PP(w)$, for every possible path, starting with every possible page, we will get exactly the PageRank value we are seeking. Given that all these probability values are positive, if we sum the probabilities for a *subset* of these paths, we get a *lower bound* on PageRank. This is exactly what the algorithms in the next section do; they enumerate every possible path during a crawl greedily from the high probability paths, and sum up their respective path probabilities.

4.2 Algorithm RankMass

In this section we finally introduce the RankMass crawling algorithm which (1) enumerates every possible path originating from the set of trusted pages, (2) compute the PageRank lower bound of each page through this enumeration and (3) greedily downloads the page with the highest lower bound. The pseudo code for the algorithm is presented in Figure 4.[3]

The algorithm maintains three important data structures: (1) $UnexploredPaths$ keeps track of the paths that have been discovered but not explored (and their path probabilities). (2) $sumPathProb_i$ maintains the sum of the path probabilities in $UnexploredPaths$ that end at p_i. That is,

$$sumPathProb_i = \sum_{w_{ji} \in UnexloredPaths} PP(w_{ji}).$$

As we will see later, for an *undownloaded* page p_i, $sumPathProb_i$ works as the PageRank lower bound during a crawl. (3) $\underline{r}_i$ stores the sum of the path probabilities to p_i that have already been explored. For a *downloaded* page p_i, $\underline{r}_i$ works as the PageRank lower bound during a crawl.

We now explain how the algorithm proceeds. The algorithm begins in lines 2–7 by initializing $UnexploredPaths$ with $\{p_i\}$'s (i.e., the paths that contain one trusted page) and their path probabilities $PP(\{p_i\}) = (1-d)t_i$. Note that $sumPathProb_i$ is set to $(1-d)t_i$ in line 7, because currently $UnexploredPaths$ contains only one path $\{p_i\}$ for each page p_i.

After this initialization, the algorithm starts exploring the paths in $UnexploredPaths$ in the loop between line 9 and 25. For now, ignore the crawling step in line 11, so that we can understand how the paths are explored and expanded.

[3]This version of the RankMass crawling algorithm includes a number of unnecessary steps to help the reader follow the code. More simplified version of the algorithm will be presented shortly.

```
 0: Variables:
 0:   $UnexploredPaths$:                                   List of unexplored paths and their path probabilities
 0:   $sumPathProb_i$:                                     Sum of the probabilities of all unexplored paths leading to $p_i$
 0:   $\underline{r}_i$:                                   Partial sum of the probability of being in $p_i$
 0:
 1: RankMassCrawl()
 2:   // Initialize:
 3:   $\underline{r}_i = 0$ for each $i$                   // Set initial probability sum to be zero.
 4:   $UnexploredPaths = \{\}$                             // Start with empty set of paths.
 5:   Foreach $(t_i > 0)$:                                 // Add initial paths of jumping to a trusted page and
 6:       Push [path: $\{p_i\}$, prob: $(1-d)t_i$] to $UnexploredPaths$   // the probability of the random jump.
 7:       $sumPathProb_i = (1-d)t_i$                       // For every trust page $p_i$, we currently have only one path $\{p_i\}$
 8:                                                        // and its path probability is $(1-d)t_i$.
 9:   While $(\sum_i \underline{r}_i < 1 - \epsilon)$:
10:       Pick $p_i$ with the largest $sumPathProb_i$.     // Get the page with highest $sumPathProb_i$.
11:       Download $p_i$ if not downloaded yet             // Crawl the page.
12:
13:       // Now expand all paths that end in $p_i$
14:       $PathsToExpand =$ Pop all paths ending with $p_i$   // Get all the paths leading to $p_i$,
15:                          from $UnexploredPaths$
16:       Foreach $p_j$ linked to from $p_i$              // and expand them by adding $p_i$'s children to the paths.
17:           Foreach $[path, prob] \in PathsToExpand$
18:               $path' = path \cdot p_j$                 // Add the child $p_j$ to the path,
19:               $prob' = \frac{d}{c_i} \cdot prob$       // compute the probability of this expanded path,
20:               Push $[path', prob']$ to $UnexploredPaths$   // and add the expanded path to $UnexploredPaths$.
21:           $sumPathProb_j = sumPathProb_j + \frac{d}{c_i} sumPathProb_i$   // Add the path probabilities of the newly added paths to $p_j$.
22:
23:       // Add the probabilities of just explored paths to $\underline{r}_i$
24:       $\underline{r}_i = \underline{r}_i + sumPathProb_i$   // We just explored all paths to $p_i$. Add their probabilities
25:       $sumPathProb_i = 0$                              // to $\underline{r}_i$.
```

Figure 4: RankMass Crawling Algorithm.

In every iteration of the loop, the page with the highest $sumPathProb_i$ is chosen in line 10. Then in lines 13–21 all paths in $UnexploredPaths$ that end in p_i are expanded to p_i's children and added back to $UnexploredPaths$. After the paths are expanded their path probabilities are added to $\underline{r}_i$ in lines 23–25.

We now state two important properties of the algorithm using the following theorems.

Theorem 4 *When the RankMass algorithm stops, the RankMass of the downloaded pages is at least $1 - \epsilon$. That is:*

$$\sum_{p_i \in D_C} r_i \geq 1 - \epsilon \qquad \square$$

Theorem 5 *While the algorithm is running, for any page $p_i \notin D_C$ (i.e., a page that has not been downloaded), $r_i \geq sumPathProb_i$. That is, $sumPathProb_i$ is the PageRank lower bound of an undownloaded page p_i.* $\square$

Theorem 4 provides the RankMass guarantee of the downloaded pages D_C when the algorithm stops. Theorem 5 confirms our earlier assertion that the algorithm gives higher priority to the pages with higher PageRank lower bounds during a crawl (to see this, look at lines 10–11 in Figure 4).

For lack of space we only provide the main ideas of the proof here. To prove Theorem 4, we have to verify the following two facts: (1) $\underline{r}_i > 0$ if and only if $p_i \in D_C$ and (2) $r_i \geq \underline{r}_i$ (i.e., $\underline{r}_i$ is the lower bound of the downloaded page p_i). The first fact is easy to verify because in Figure 4, $\underline{r}_i$ is increased (line 24) only after p_i is downloaded (line 11). To verify the second fact, all that is left to show is that (1) $\underline{r}_i$ values are essentially the sums of the path probability $PP(w)$ for the paths w that lead to p_i and (2) no such path w is counted twice. Similarly, to prove Theorem 5, we have to show that (1) $sumPathProb_i$ values are essentially sums of $PP(w)$'s for the paths w that lead to p_i and (2) no such path w is counted twice. By going over the steps of the algorithm carefully, it is relatively straightforward to verify these facts, which proves the above theorems.

Note that the page p_i chosen at line 10 of the algorithm may have already been downloaded. In this case, the purpose of the lines 11 through 25 is not to redownload the page, but to expand the paths in order to improve the PageRank lower bound. This "virtual crawl" or "internal probability propagation" helps the RankMass crawler provide a tighter RankMass lower bound, but may require a random access to disk since the links from p_i may not reside in main memory. In the next section, we will briefly talk about how we may minimize the number of these random accesses by "batching" the page download and path-expansion steps.

Before we finish our discussion on the RankMass crawling algorithm, we finally discuss how we can simplify the algorithm in Figure 4. For this simplification, we first note that the information in $UnexploredPaths$ does not have any effect over the control flow of the algorithm. That is, even if we remove lines 4, 6, 14–15 and 17–20, and do not maintain $UnexploredPaths$ at all, the algorithm still proceeds the same way. Second, we note that computing individual $\underline{r}_i$ values for every page p_i is an overkill. All we need is the *sum* of $\underline{r}_i$ values, so that we can compute the RankMass lower bound of the downloaded pages. Therefore, instead of keeping one $\underline{r}_i$ for every page p_i, we can just keep the sum of $\underline{r}_i$ values in a single variable CRM. Based on this discussion, we show our final simplified RankMass crawling algorithm in Figure 5.[4]

4.3 Windowed-RankMass Crawler

[4]In the algorithm, we also renamed $sumPathProb_i$ with rm_i for conciseness.

```
0: Variables:
0:   $CRM$: RankMass lower bound of crawled pages
0:   $rm_i$: Lower bound of PageRank of $p_i$.
0:
1: RankMassCrawl()
2:   $CRM = 0$
3:   $rm_i = (1 - d)t_i$ for each $t_i > 0$
4:   While ($CRM < 1 - \epsilon$):
5:     Pick $p_i$ with the largest $rm_i$.
6:     Download $p_i$ if not downloaded yet
7:     $CRM = CRM + rm_i$
8:     Foreach $p_j$ linked to by $p_i$:
9:       $rm_j = rm_j + \frac{d}{c_i} rm_i$
10:    $rm_i = 0$
```

Figure 5: RankMass Crawling Algorithm.

The RankMass algorithm presented in the previous section is an extremely greedy algorithm that attempts to not only download the page with the highest PageRank early, but attempts to provide us with a tight guarantee of the RankMass collected by the crawl. This type of greediness increases our chances of being close to the optimal crawl, but the RankMass algorithm requires random access to the graph structure, and random access to storage is inherently expensive.

In this section, we present an algorithm which allows us to adjust its greediness through a special *window* parameter. As we will see, setting the window to 100% will give us the L-Neighbor algorithm, and as the parameter approaches 0%, the algorithm behaves more and more like the RankMass crawler.

The Windowed-RankMass algorithm is an adaptation of the RankMass algorithm and is designed to allow us to reduce the overhead by batching together sets of probability calculations and downloading sets of pages at a time. Batching together the downloads may lead to page downloads that are less close to the optimal. For example, consider the last downloaded page of a batch of pages. All the pages downloaded so far in the batch, could have been used to further tighten the PageRank lower bounds, and thus increase the probability of downloading a page with a higher PageRank value. On the other hand, batching the downloads together also allows us to batch together the probability calculations which in turn makes it computationally viable to access the graph structure through serial access to the disk rather than the inherently slow random access. The algorithm itself is presented in Figure 6 and is very similar to the RankMass algorithm that was presented in Figure 5.

There are two key differences between this algorithm and the RankMass algorithm. The first difference is that the top *window*% of the pages are crawled in line 7, downloaded as a batch, rather than just the page with the highest rm_i. The second difference is that the probability calculations in line 8–12 are done for all pages downloaded so far rather than just the page with the highest rm_i. The stopping condition in line 6 is identical to the one used in the previous algorithm.

Windowed-RankMass algorithm is actually a family of crawling algorithms that allows us to decide exactly how greedy we want to be. The appropriate window size can be chosen according to the needs, purpose and resources available for the crawl.

In the next section we evaluate the three algorithms we introduced in this paper.

5. EXPERIMENTAL RESULTS

```
0: Variables:
1:   $CRM$:
2:   $rm_i$:
3:
4: Crawl()
5:   $rm_i = (1 - d)t_i$ for each $t_i > 0$
6:   While ($CRM < 1 - \epsilon$):
7:     Download top $window$% pages according to $rm_i$
8:     Foreach page $p_i \in D_C$
9:       $CRM = CRM + rm_i$
10:      Foreach $p_j$ linked to by $p_i$:
11:        $rm_j = rm_j + \frac{d}{c_i} rm_i$
12:      $rm_i = 0$
```

Figure 6: Windowed-RankMass Crawling Algorithm.

In this paper, we introduced three new crawling algorithms: L-Neighbor, Windowed-RankMass, and RankMass. The main difference between these three algorithms is the greediness at which they attempt to download the page with the highest PageRank. In this section, we first discuss the metrics we will use to compare and evaluate these algorithms. Next, we will describe the experimental setup and data used, and finally, we will introduce the experimental results.

Metrics of Evaluation There are three metrics we will use to evaluate each algorithm: 1. How much RankMass is collected during the crawl, 2. How much RankMass is "known" to have been collected during the crawl, and 3. How much computational and performance overhead the algorithm introduces. The first metric we use to evaluate these algorithms is the actual RankMass they collect in the first X pages downloaded. This measure indicates how well the algorithm actually performs under the RankMass coverage metric defined in Section 2.

However, having downloaded a certain percent of the Web's RankMass does not mean that we *know* we can stop the crawl. Unless the algorithm provides a guarantee that the RankMass downloaded is above our stopping condition ϵ, we cannot stop the crawl. Therefore, the second metric by which we evaluate the crawling algorithms is the RankMass guaranteed to have been downloaded after having downloaded X pages. The closer the guarantee is to the actual RankMass collected, the less time we spend on "unnecessarily" downloading pages.

Finally, the third metric attempts to identify the overhead of collecting the RankMass early and providing the lower bound. In other words, this metric completes the evaluation by saying you have to invest this much more computing time in order to achieve this much benefit.

Experiment setup We simulate the algorithms L-Neighbor, RankMass, and Windowed-Rankmass on a subgraph of the Web of 141 million URLs. This subgraph of the Web includes HTML files only, obtained from a crawl that was performed between the dates December of 2003 and January of 2004. The 141 millon URLs span over 6.9 million host names and 233 top level domains. Parsing the HTML files and extracting the links was done using a simple HTML parser. Only the links starting with "http://" were retained, and the only URL normalization employed was the removal of terminating slashes.

In order to compute the actual RankMass collected in the algorithms, we needed to calculate the PageRank values for all the pages of our subgraph. Although PageRank is defined over the entire web, calculating it over the entire infinite web is impossible. Therefore we calculated the PageRank values over the subgraph of the web described above. Additionally, in order to compute the PageRank values of the pages, we have to assume a particular trust vector. In all the simulations, unless explicitly specified otherwise, we assume uniform trust distribution for the 159 trusted set of pages used in the TrustRank paper [16]. [5]

Simulations of the algorithms were run on machines with four Gigabytes of memory, and four 2.4 GHz CPUs each. However, we did not employ any parallel programming techniques in our code, so we did not take advantage of more than one CPU at a time. The algorithm simulations were implemented in C++ and compiled using the g++ compiler without any optimizations. Thanks to the large memory of the machines, most of the processing was done in memory. The only disk access that was done in L-Neighbor and Windowed-RankMass was serial disk access to the graph structure stored on disk. In the RankMass algorithm, which requires random access to the graph structure, the graph structure was accessed through an SQlite [1] database.

Next we present the results of this evaluation for the three algorithms against the first two metrics: actual RankMass collected and RankMass lower bound guarantee.

L-Neighbor Algorithm In Figure 7(a) we can see the simulation results for the L-Neighbor crawling algorithm. The horizontal axis represents the number of documents downloaded so far in the crawl while the vertical axis represents the RankMass of these documents. Whether it is the actual RankMass or guaranteed RankMass depends on the specific curve. The optimal curve shows the actual RankMass collected by an ideal crawler. An ideal crawler in this case tries to maximize the RankMass collected in each download and is assumed to know all the URLs and the PageRank value of the pages they point to. Thus, an ideal crawler would simply download the uncrawled page that has the highest PageRank value.

For the L-Neighbor algorithm we can see two values in this graph, the actual RankMass collected and the "guarantee", the lower bound on the RankMass collected. The guarantee is calculated at the time of the crawl, so it is known during the crawl and can help us know when to stop. On the other hand, the actual RankMass collected is calculated after the crawl is complete, by calculating the PageRank values for all the pages downloaded, and summing them up in the download order. As expected, the optimal curve is higher than the actual curve which is higher than the lower bound guarantee.

The L-Neighbor RankMass guarantee starts off considerably below the actual RankMass collected, and gets tighter only in an advanced stage of the crawl. The algorithm downloads more than 7 million pages before the guaranteed RankMass passes 0.98, while the optimal does the same with as few as 27,101 pages. However, although it takes 7 million pages for us to know we downloaded more than 0.98 RankMass, the actual RankMass collected is above 0.98 after as few as 65,000 downloads. Thus although the L-Neighbor algorithm performs very well – the actual RankMass collected is very close to optimal after as few as one million downloads – as a result of the lower bound not being tight,

[5]The paper in fact used 178 URLs, within our data set only 159 of them appeared

Algorithm	Downloads required for above 0.98% guaranteed RankMass	Downloads required for above 0.98% actual RankMass
L-Neighbor	7 million	65,000
RankMass	131,072	27,939
Windowed-RankMass	217,918	30,826
Optimal	27,101	27,101

Table 2: Downloads required for above 0.98% **RankMass**

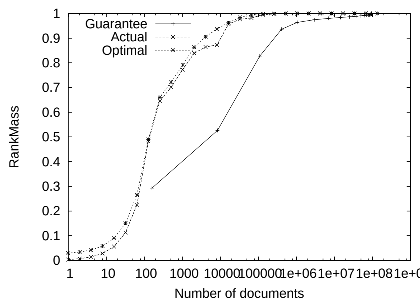
we may download more pages than necessary to meet our desired stopping condition. Given the fact that our RankMass guarantee is independent of the global Web link structure, it is very promising that we get more than 98% of the RankMass guarantee just after 7 million downloads.

RankMass Algorithm Figure 7(c) shows the simulation results for the RankMass crawling algorithm. The optimal curve shown is the same as described before as are the vertical axis and horizontal axis.

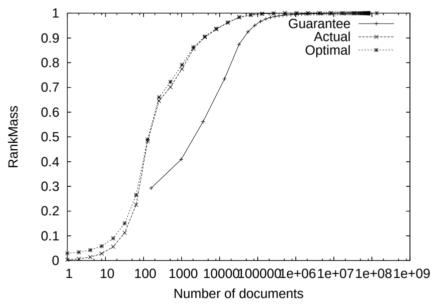
Unlike in the L-neighbor case the guarantee provided during the run of the RankMass algorithm is very tight. This means that a crawl will be able to stop earlier in the RankMass algorithm, when run with the same stop condition ϵ. For example, the first 131,072 pages downloaded already give a guaranteed RankMass above 0.98, and an actual RankMass of above 0.98 is achieved after 27,939 page downloads. Additionally, note that both the guaranteed RankMass and the actual RankMass collected are very close to the optimal, even at an early stage, and become even more tight as the crawl progresses. For example, the first 262,144 pages downloaded by the RankMass crawler give us a guaranteed RankMass of 0.996 and an actual RankMass of 0.999, while the optimal crawler gives a 0.999 RankMass as well.

Windowed-RankMass Algorithm Figure 7(b) shows us the RankMass collected using the Windowed-RankMass algorithm with a window size of 10%. The optimal curve is again the same, as are the horizontal and vertical axes. When compared to the L-Neighbor algorithm, both the guaranteed RankMass and the actual RankMass collected is higher. For example, Windowed-RankMass requires only 217,918 pages to achieve a guaranteed RankMass of 0.98 and only 30826 pages to achieve the same in actual RankMass. However, the RankMass crawler performs better than both of the other algorithms, under both the guaranteed and actual RankMass metrics. For this simulation we arbitrarily chose a window size of 10%. What are the effects of different window sizes on the actual and guaranteed RankMass collected?

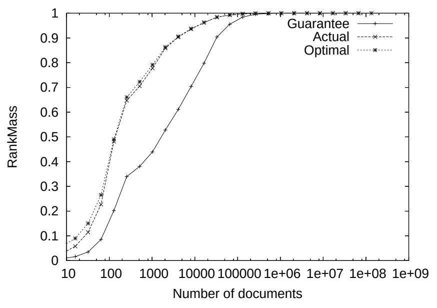
Window Size Figure 8 shows us the actual RankMass collected per documents fetched for four window sizes. Note that when the window size is set to 100% then the windowed RankMass algorithm behaves just like L-Neighbor. The difference between a window size of 100% and the window sizes of 20%, 10%, and 5% is readily apparent. The algorithm with the larger window size of 100%, although performing quite well, takes longer to come close to the optimal value. Window sizes of 5%, 10% and even 20% give results that are very close to the peformance of RankMass and the optimal crawler. Also worth noting is the stability of the curve, while RankMass, and the windowed Rankmass with a win-



(a) L-Neighbor Crawler



(b) 10%-Windowed RankMass Crawler



(c) RankMass Crawler

Figure 7: RankMass vs Number of Documents for Three Algorithms

dow of 5%, 10% and 20% show relatively smooth curves, the *L*-Neighbor shows some perturbations. The implication is that a smaller window size and the RankMass crawling algorithm will give better and more consistent results.

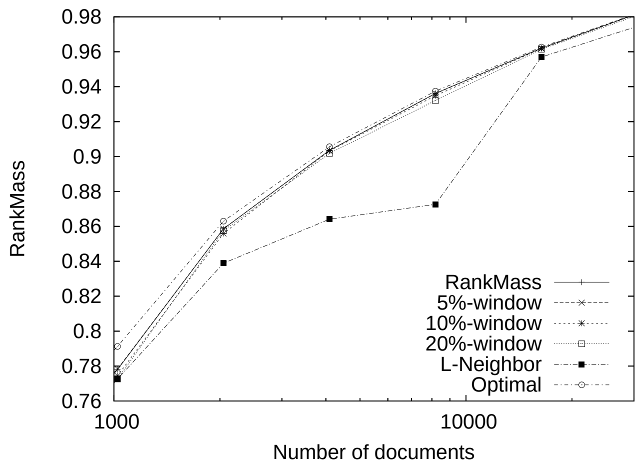


Figure 8: Examining Window size in Windowed RankMass Crawler

The next question we wish to answer now is what price we pay for this added benefit and if so, how high.

Running Time In order to compare the performance of the three algorithms, we ran each algorithm simulation and noted the time and the number of iterations needed to download 80 million pages. The results are summarized in Table 3. Examining the results for 20%-windowed, 10%-windowed and 5%-windowed we can see that halving the window size doubles the number of iterations and is close to doubling the running time. *L*-Neighbor is clearly close to three times faster than 20%-windowed. Finally, RankMass is considerably slower than all of them, according to an initial examination it is mainly due to the random access to the graphs structure. Given these results, we find that a 20% window may be a reasonable choice for the window size, since it gives roughly equivalent RankMass performance at a significantly lower computational cost.

Window	Hours	number of iterations	Number of documents
L-Neighbor	1:27	13	83,638,834
20%-windowed	4:39	44	80,622,045
10%-windowed	10:27	85	80,291,078
5%-Windowed	17:52	167	80,139,289
RankMass	25:39[6]	*not comparable*	10,350,000

Table 3: Algorithm performance by iterations and running time

How many seeds should be used? Now we investigate the issue of seed set selection, when the crawling cannot start from all trusted pages, for example when the trust is distributed between an unbounded number of pages. In this case, one important decision people who run crawlers need to make is deciding how many pages to use to seed the crawl. Deciding on the number of seeds is influenced by many factors such as: the connectivity of the web, spammability of the search engine, efficiency of the crawl, and many other factors. We now examine how the guarantee on the RankMass and the actual RankMass collected are affected by the number of seeds.

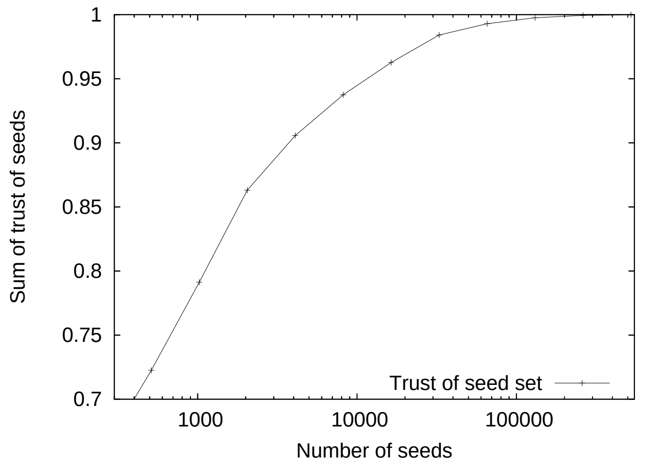


Figure 9: Overall trust of seeds vs number of seeds

In this experiment we assume that the trust is distributed similarly to the way PageRank is distributed. More pre-

[6]By the time the RankMass algorithm downloaded 10,350,000 pages, it has already taken a significantly longer time than any other algorithms, so we stopped the RankMass crawling at that point. With extrapolation, we estimate that it would have taken at least 222 for the RankMass crawler to have downloaded 80,000,000 pages.

cisely, as the trust score of each page, we use its PageRank value under the original PageRank definition (i.e., the values computed with the *uniform* trust distribution over all pages). Since PageRank is known to be distributed by power-law under the original definition, the trust follows the power-law curve. Choosing any subset of the pages as the seed set will result in only part of the trust being included according to Theorem 3, so it make sense to choose the pages with the highest trust values as your seed to maximize the RankMass guarantee. Figure 9 plots the overall trust of the seeds as a function of the number of seeds. The graph shows that if we use 100,000 most trusted pages as the seed set (less than 0.1% of our dataset), their trust sum is more than 99% of all trust, so the RankMass guarantee provided from this seed set will decrease by less than 1% due to a finite number of seed-set selection.

To investigate the impact of the seed set size on RankMass guarantee further, we ran the L-Neighbor algorithm with three seed set sizes: 160, 1600, and 16000, each time choosing the pages with the highest trust values as seeds. The larger the seed set is, the higher its trust value sum is, and the higher guarantee it can give on RankMass. On the other hand, the large seed sets are likely to include pages with small trust values whose neighbors are less likely to have high PageRank. In Figure 10 we plot the results of this experiment.

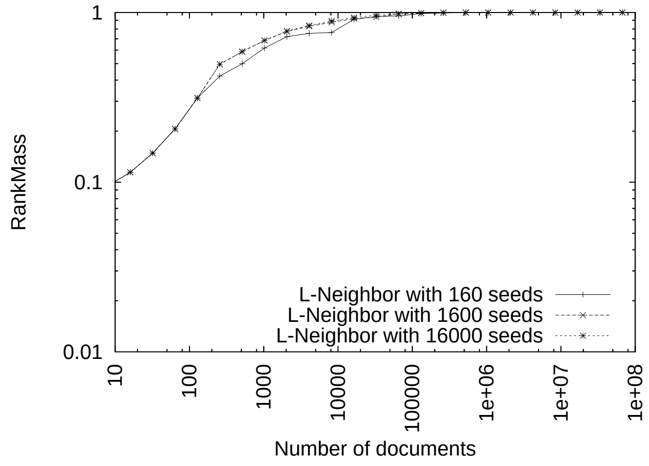
In Figure 10(b) we plot the guaranteed RankMass as a function of the number of documents downloaded. As we can see, the difference between the guaranteed RankMass collected using the different seeds sets, quickly becomes very apparent. The reason for this is again, the different trust captured in the different seed sets. This demonstrates the advantage of using larger seed sets that capture more trust. However, as we can see in Figure 10(a), when comparing the actual RankMass collected, while during the first few downloads, the actual RankMass collected is larger when the seed set is larger, any clear advantage one way or another is not apparent in the more advanced stages of the crawl. Thus, when crawling using the L-Neighbor algorithm, a larger seed set that includes more trust affects the guarantee very strongly while affecting the actual RankMass collected only in the early stages of the crawl.

6. RELATED WORK

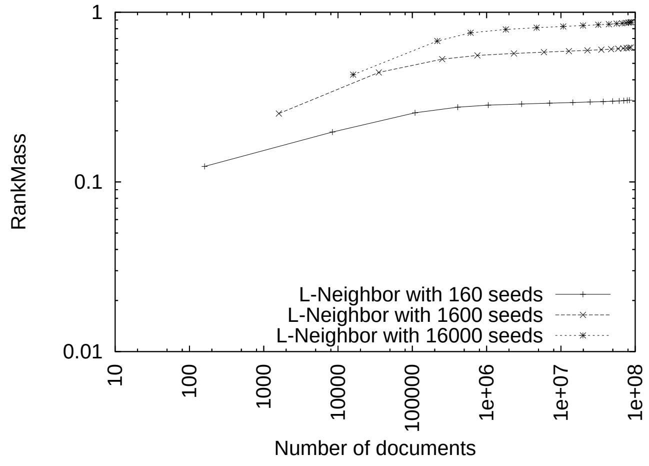
We now briefly go over the related work.

Crawling has been the subject of extensive research [24, 27, 14, 10, 13] and Different aspects of Web crawling has been studied before. For example, in [29] Wolf et al. discuss re-crawling strategies in order to maintain the freshness of the search engine's index. In this section we go over the work that is most relevant to ours.

URL prioritization for crawling There exists a large body of work on URL prioritization for Web crawling, which is the main topic of this paper. Instances of our family of algorithms have already been described and evaluated before. For example, our L-Neighbor algorithm essentially follows the well-known breadth-first crawling strategy, which has been shown to perform well empirically in [24, 12]. However, as far as we know, due to the fundamental difficulty of computing the exact PageRank from an incomplete subset of the Web, most of the previous studies were experimental or focused on the eventual convergence of the algorithms [24, 12, 2, 4]. As far as we know, our work is the first that shows analytically why some of these algorithms perform well and provide a tight guarantee on the PageRank coverage of the downloaded pages.



(a) Actual RankMass collected by L-Neighbor with different seed set sizes



(b) Guaranteed RankMass collected by L-Neighbor with different seed set sizes

Figure 10: RankMass collected by L-Neighbor with different seed set sizes

A different type of crawler is the focused crawler which sets to download pages related to a specific topic. The basic premise of focused crawling is that pages pointed to by a page on a specific topic are more likely to be related to that topic. In [10] Chakrabarti et al. are first to introduce such a crawler, and an accelerated approach is presented in [11]. In [15] Ester et al. examine a two level approach to focused crawling, finding relevant websites first and then crawling inside them. Although this work is similar to our own, the important difference is that the importance of pages is *defined* by the links to them. This difference allows us to provide the upper bounds we provide in this paper. Future work may attempt to tackle focused crawling using topic sensitive PageRank [17].

Search engine index quality Other related papers focus on the quality of pages covered by a search engine's index and the pages' interest to users. In [19] Heydon et al. are first to introduce a measure for the quality of a search engine's index. This is the measure we use in this paper which we refer to as RankMass. The authors describe a technique which uses random walks to estimate the RankMass of a search engine's index. However, this paper does not discuss upper bounds and does not define a crawling scheme that sets to download higher quality documents earlier in the crawl.

In [3], Baeza-Yates et al. analyze how deep a user is likely to browse inside a single Web site with an unbounded number of pages. The paper presents several random-surfer models that are used to analyze the level a user reaches inside the web site. The authors backup their findings with empirical data from Web logs and by summing the PageRank values of the first levels of a specific site. However, the paper focuses on empirical analysis of the interestingness of a Web site's pages while our paper, we focus on developing new crawling algorithms.

In [25] Pandey and Olston define their own metric of the quality of a search engine's index and optimize the *recrawling* of pages to maximize this quality. However, this work does not tackle the task of optimizing the crawl of the unknown, undownloaded part of the web as we do.

The PageRank metric PageRank has been the subject of extensive research since its introduction in [8]. The general version of PageRank, which is also called personalized PageRank, is discussed in [20, 16, 17]. In [20, 21] Jeh and Widom tackle calculating many different PageRank vectors with different trust vectors efficiently. In [16] Gyongyi et al. use the trust vector to combat spam. In [17] topic specific PageRank vectors are calculated and in [30] topic specific TrustRank values are calculated to combat spam. In [22] Haveliwala et al. present an accelerated method of calculating PageRank. Abiteboul et al. present an online adaptive page importance computation algorithm, which is a slight variation of the PageRank metric, and discusses its application in the context of a Web crawler. In [28, 26] PageRank is approximated in a distributed peer-to-peer environment. In [6] Becchetti and Castillo explore the connection between the damping factor used and the PageRank distribution. Many other papers such as [7, 23] analyze PageRank extensively, however, we are not aware of previous work that developed the upper bounds and the crawling algorithms as they are presented here.

7. CONCLUSION

In this paper we investigated a family of crawling algorithms that range from a lightweight and effective breadth-first-search-style algorithm, up to the very aggressively greedy RankMass algorithm, which sets to maximize the PageRank of every page downloaded, and gives close to optimal results. All these algorithms provide a guarantee of the PageRank collected during and at the end of the crawl, despite having access to only a subgraph of the Web at any time. This guarantee allows the search engine to specify a formal goal-driven stopping condition, thus continuing the crawl only long enough to download the desired percent of the Web's PageRank.

8. REFERENCES

[1] Sqlite database engine. http://www.sqlite.org/.

[2] S. Abiteboul, M. Preda, and G. Cobena. Adaptive on-line page importance computation. In *Proc. 12th WWW*, pages 280–290, 2003.

[3] R. Baeza-Yates and C. Castillo. Crawling the infinite Web: five levels are enough. In *Proc. 3rd WAW*, Rome, Italy, October 2004. Springer LNCS.

[4] R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez. Crawling a country: better strategies than breadth-first for web page ordering. In *Proc. 14th WWW*, pages 864–872. ACM Press New York, NY, USA, 2005.

[5] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's index. In *Proc. 15th WWW*, pages 367–376, 2006.

[6] L. Becchetti and C. Castillo. The distribution of pagerank follows a power-law only for particular values of the damping factor. In *Proc. 15th WWW*, pages 941–942, 2006.

[7] M. Bianchini, M. Gori, and F. Scarselli. Inside PageRank. *ACM TOIT*, 5(1):92–128, 2005.

[8] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Proc. 7th WWW*, 30(1-7):107–117, 1998.

[9] M. Brinkmeier. PageRank revisited. *ACM TOIT*, 6(3):282–301, 2006.

[10] S. Chakrabarti, M. van den Berg, and B. Dom. Focused Crawling: A New Approach for Topic-Specific Resource Discovery. In *Proc. 8th WWW*, 1999.

[11] Soumen Chakrabarti, Kunal Punera, and Mallela Subramanyam. Accelerated focused crawling through online relevance feedback. In *Proc. 11th WWW*, pages 148–159, 2002.

[12] J. Cho, H. Garcia-Molina, and L. Page. Efficient Crawling Through URL Ordering. *Proc. 7th WWW*, 30(1-7):161–172, 1998.

[13] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. In *Proc. 11th WWW*, pages 124–135, 2002.

[14] M. Diligenti, F. Coetzee, S. Lawrence, C.L. Giles, and M. Gori. Focused crawling using context graphs. In *Proc. 26th VLDB*, pages 527–534. September, 2000.

[15] Martin Ester, Hans-Peter Kriegel, and Matthias Schubert. Accurate and efficient crawling for relevant websites. In *Proc. 30th VLDB*, pages 396–407, 2004.

[16] Z. Gyongyi, H. Garcia-Molina, and J. Pedersen. Combating web spam with TrustRank. 2004.

[17] Taher H. Haveliwala. Topic-sensitive pagerank. In *Proc. 11th WWW*, Honolulu, Hawaii, May 2002.

[18] T.H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE TKDE*, 15(4):784–796, 2003.

[19] M.R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. Measuring index quality using random walks on the Web. *Proc. 8th WWW*, 31(11-16):1291–1303, 1999.

[20] G. Jeh and J. Widom. Scaling personalized web search. Technical report, Stanford University, 2002.

[21] G. Jeh and J. Widom. Scaling personalized web search. In *Proc. 12th WWW*, pages 271–279, 2003.

[22] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating pagerank computations. In *Proc. 12th WWW*, pages 261–270, 2003.

[23] A. N. Langville and C. D. Meyer. Deeper Inside PageRank. *Internet Mathematics*, 1(3):335–380, 2005.

[24] M. Najork and J. L. Wiener. Breadth-first crawling yields high-quality pages. *Proc. 10th WWW*, pages 114–118, 2001.

[25] S. Pandey and C. Olston. User-centric web crawling. In *Proc. 14th WWW*, pages 401–411, 2005.

[26] J. X. Parreira, D. Donato, S. Michel, and G. Weikum. Efficient and decentralized pagerank approximation in a peer-to-peer web search network. In *Proc. 32nd VLDB*, pages 415–426, 2006.

[27] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. *Proc. 27th VLDB*, pages 129–138, 2001.

[28] Y. Wang and D. J. DeWitt. Computing pagerank in a distributed internet search engine system. In *Proc. 30th VLDB*, pages 420–431, 2004.

[29] J. L. Wolf, M. S. Squillante, P. S. Yu, J. Sethuraman, and L. Ozsen. Optimal crawling strategies for web search engines. In *Proc. 11th WWW*, pages 136–147, 2002.

[30] B. Wu, V. Goel, and B. D. Davison. Topical trustrank: using topicality to combat web spam. In *Proc. 15th WWW*, pages 63–72, 2006.