Chapter 5

Link Analysis

One of the biggest changes in our lives in the decade following the turn of the century was the availability of efficient and accurate Web search, through search engines such as Google. While Google was not the first search engine, it was the first able to defeat the spammers who had made search almost useless. Moreover, the innovation provided by Google was a nontrivial technological advance, called "PageRank." We shall begin the chapter by explaining what PageRank is and how it is computed efficiently.

Yet the war between those who want to make the Web useful and those who would exploit it for their own purposes is never over. When PageRank was established as an essential technique for a search engine, spammers invented ways to manipulate the PageRank of a Web page, often called link spam.[1] That development led to the response of TrustRank and other techniques for preventing spammers from attacking PageRank. We shall discuss TrustRank and other approaches to detecting link spam.

Finally, this chapter also covers some variations on PageRank. These techniques include topic-sensitive PageRank (which can also be adapted for combating link spam) and the HITS, or "hubs and authorities" approach to evaluating pages on the Web.

5.1 PageRank

We begin with a portion of the history of search engines, in order to motivate the definition of PageRank,[2] a tool for evaluating the importance of Web pages in a way that it is not easy to fool. We introduce the idea of "random surfers," to explain why PageRank is effective. We then introduce the technique of "taxation" or recycling of random surfers, in order to avoid certain Web structures

[1] Link spammers sometimes try to make their unethicality less apparent by referring to what they do as "search-engine optimization."

[2] The term PageRank comes from Larry Page, the inventor of the idea and a founder of Google.

that present problems for the simple version of PageRank.

5.1.1 Early Search Engines and Term Spam

There were many search engines before Google. Largely, they worked by crawling the Web and listing the *terms* (words or other strings of characters other than white space) found in each page, in an inverted index. An *inverted index* is a data structure that makes it easy, given a term, to find (pointers to) all the places where that term occurs.

When a *search query* (list of terms) was issued, the pages with those terms were extracted from the inverted index and ranked in a way that reflected the use of the terms within the page. Thus, presence of a term in a header of the page made the page more relevant than would the presence of the term in ordinary text, and large numbers of occurrences of the term would add to the assumed relevance of the page for the search query.

As people began to use search engines to find their way around the Web, unethical people saw the opportunity to fool search engines into leading people to their page. Thus, if you were selling shirts on the Web, all you cared about was that people would see your page, regardless of what they were looking for. Thus, you could add a term like "movie" to your page, and do it thousands of times, so a search engine would think you were a terribly important page about movies. When a user issued a search query with the term "movie," the search engine would list your page first. To prevent the thousands of occurrences of "movie" from appearing on your page, you could give it the same color as the background. And if simply adding "movie" to your page didn't do the trick, then you could go to the search engine, give it the query "movie," and see what page *did* come back as the first choice. Then, copy that page into your own, again using the background color to make it invisible.

Techniques for fooling search engines into believing your page is about something it is not, are called *term spam*. The ability of term spammers to operate so easily rendered early search engines almost useless. To combat term spam, Google introduced two innovations:

1. PageRank was used to simulate where Web surfers, starting at a random page, would tend to congregate if they followed randomly chosen outlinks from the page at which they were currently located, and this process were allowed to iterate many times. Pages that would have a large number of surfers were considered more "important" than pages that would rarely be visited. Google prefers important pages to unimportant pages when deciding which pages to show first in response to a search query.

2. The content of a page was judged not only by the terms appearing on that page, but by the terms used in or near the links to that page. Note that while it is easy for a spammer to add false terms to a page they control, they cannot as easily get false terms added to the pages that link to their own page, if they do not control those pages.

Simplified PageRank Doesn't Work

As we shall see, computing PageRank by simulating random surfers is a time-consuming process. One might think that simply counting the number of in-links for each page would be a good approximation to where random surfers would wind up. However, if that is all we did, then the hypothetical shirt-seller could simply create a "spam farm" of a million pages, each of which linked to his shirt page. Then, the shirt page looks very important indeed, and a search engine would be fooled.

These two techniques together make it very hard for the hypothetical shirt vendor to fool Google. While the shirt-seller can still add "movie" to his page, the fact that Google believed what other pages say about him, over what he says about himself would negate the use of false terms. The obvious countermeasure is for the shirt seller to create many pages of his own, and link to his shirt-selling page with a link that says "movie." But those pages would not be given much importance by PageRank, since other pages would not link to them. The shirt-seller could create many links among his own pages, but none of these pages would get much importance according to the PageRank algorithm, and therefore, he still would not be able to fool Google into thinking his page was about movies.

It is reasonable to ask why simulation of random surfers should allow us to approximate the intuitive notion of the "importance" of pages. There are two related motivations that inspired this approach.

- Users of the Web "vote with their feet." They tend to place links to pages they think are good or useful pages to look at, rather than bad or useless pages.

- The behavior of a random surfer indicates which pages users of the Web are likely to visit. Users are more likely to visit useful pages than useless pages.

But regardless of the reason, the PageRank measure has been proved empirically to work, and so we shall study in detail how it is computed.

5.1.2 Definition of PageRank

PageRank is a function that assigns a real number to each page in the Web (or at least to that portion of the Web that has been crawled and its links discovered). The intent is that the higher the PageRank of a page, the more "important" it is. There is not one fixed algorithm for assignment of PageRank, and in fact variations on the basic idea can alter the relative PageRank of any two pages. We begin by defining the basic, idealized PageRank, and follow it by modifications that are necessary for dealing with some real-world problems concerning the structure of the Web.

Think of the Web as a directed graph, where pages are the nodes, and there is an arc from page p_1 to page p_2 if there are one or more links from p_1 to p_2. Figure 5.1 is an example of a tiny version of the Web, where there are only four pages. Page A has links to each of the other three pages; page B has links to A and D only; page C has a link only to A, and page D has links to B and C only.

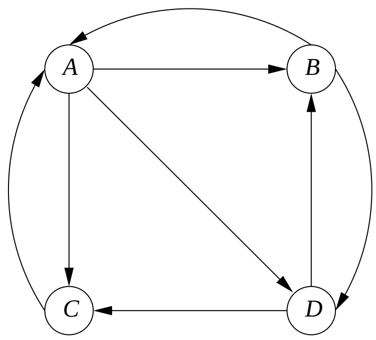

Figure 5.1: A hypothetical example of the Web

Suppose a random surfer starts at page A in Fig. 5.1. There are links to B, C, and D, so this surfer will next be at each of those pages with probability 1/3, and has zero probability of being at A. A random surfer at B has, at the next step, probability 1/2 of being at A, 1/2 of being at D, and 0 of being at B or C.

In general, we can define the *transition matrix of the Web* to describe what happens to random surfers after one step. This matrix M has n rows and columns, if there are n pages. The element m_{ij} in row i and column j has value $1/k$ if page j has k arcs out, and one of them is to page i. Otherwise, $m_{ij} = 0$.

Example 5.1: The transition matrix for the Web of Fig. 5.1 is

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

In this matrix, the order of the pages is the natural one, A, B, C, and D. Thus, the first column expresses the fact, already discussed, that a surfer at A has a 1/3 probability of next being at each of the other pages. The second column expresses the fact that a surfer at B has a 1/2 probability of being next at A and the same of being at D. The third column says a surfer at C is certain to be at A next. The last column says a surfer at D has a 1/2 probability of being next at B and the same at C. □

The probability distribution for the location of a random surfer can be described by a column vector whose jth component is the probability that the surfer is at page j. This probability is the (idealized) *PageRank* function.

Suppose we start a random surfer at any of the n pages of the Web with equal probability. Then the initial vector $\mathbf{v}_0$ will have $1/n$ for each component. If M is the transition matrix of the Web, then after one step, the distribution of the surfer will be $M\mathbf{v}_0$, after two steps it will be $M(M\mathbf{v}_0) = M^2\mathbf{v}_0$, and so on. In general, multiplying the initial vector $\mathbf{v}_0$ by M a total of i times will give us the distribution of the surfer after i steps.

To see why multiplying a distribution vector $\mathbf{v}$ by M gives the distribution $\mathbf{x} = M\mathbf{v}$ at the next step, we reason as follows. The probability $\mathbf{x}_i$ that a random surfer will be at node i at the next step, is $\sum_j m_{ij}\mathbf{v}_j$. Here, m_{ij} is the probability that a surfer at node j will move to node i at the next step (often 0 because there is no link from j to i), and $\mathbf{v}_j$ is the probability that the surfer was at node j at the previous step.

This sort of behavior is an example of the ancient theory of *Markov processes*. It is known that the distribution of the surfer approaches a limiting distribution $\mathbf{v}$ that satisfies $\mathbf{v} = M\mathbf{v}$, provided two conditions are met:

1. The graph is *strongly connected*; that is, it is possible to get from any node to any other node.

2. There are no *dead ends*: nodes that have no arcs out.

Note that Fig. 5.1 satisfies both these conditions.

The limit is reached when multiplying the distribution by M another time does not change the distribution. In other terms, the limiting $\mathbf{v}$ is an eigenvector of M (an *eigenvector* of a matrix M is a vector $\mathbf{v}$ that satisfies $\mathbf{v} = \lambda M\mathbf{v}$ for some constant *eigenvalue* λ). In fact, because M is *stochastic*, meaning that its columns each add up to 1, $\mathbf{v}$ is the *principal* eigenvector (its associated eigenvalue is the largest of all eigenvalues). Note also that, because M is stochastic, the eigenvalue associated with the principal eigenvector is 1.

The principal eigenvector of M tells us where the surfer is most likely to be after a long time. Recall that the intuition behind PageRank is that the more likely a surfer is to be at a page, the more important the page is. We can compute the principal eigenvector of M by starting with the initial vector $\mathbf{v}_0$ and multiplying by M some number of times, until the vector we get shows little change at each round. In practice, for the Web itself, 50–75 iterations are sufficient to converge to within the error limits of double-precision arithmetic.

Example 5.2: Suppose we apply the process described above to the matrix M from Example 5.1. Since there are four nodes, the initial vector $\mathbf{v}_0$ has four components, each 1/4. The sequence of approximations to the limit that we

Solving Linear Equations

If you look at the 4-node "Web" of Example 5.2, you might think that the way to solve the equation $\mathbf{v} = M\mathbf{v}$ is by Gaussian elimination. Indeed, in that example, we argued what the limit would be essentially by doing so. However, in realistic examples, where there are tens or hundreds of billions of nodes, Gaussian elimination is not feasible. The reason is that Gaussian elimination takes time that is cubic in the number of equations. Thus, the only way to solve equations on this scale is to iterate as we have suggested. Even that iteration is quadratic at each round, but we can speed it up by taking advantage of the fact that the matrix M is very sparse; there are on average about ten links per page, i.e., ten nonzero entries per column.

Moreover, there is another difference between PageRank calculation and solving linear equations. The equation $\mathbf{v} = M\mathbf{v}$ has an infinite number of solutions, since we can take any solution $\mathbf{v}$, multiply its components by any fixed constant c, and get another solution to the same equation. When we include the constraint that the sum of the components is 1, as we have done, then we get a unique solution.

get by multiplying at each step by M is:

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}, \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}, \begin{bmatrix} 15/48 \\ 11/48 \\ 11/48 \\ 11/48 \end{bmatrix}, \begin{bmatrix} 11/32 \\ 7/32 \\ 7/32 \\ 7/32 \end{bmatrix}, \ldots, \begin{bmatrix} 3/9 \\ 2/9 \\ 2/9 \\ 2/9 \end{bmatrix}$$

Notice that in this example, the probabilities for B, C, and D remain the same. It is easy to see that B and C must always have the same values at any iteration, because their rows in M are identical. To show that their values are also the same as the value for D, an inductive proof works, and we leave it as an exercise. Given that the last three values of the limiting vector must be the same, it is easy to discover the limit of the above sequence. The first row of M tells us that the probability of A must be 3/2 the other probabilities, so the limit has the probability of A equal to 3/9, or 1/3, while the probability for the other three nodes is 2/9.

This difference in probability is not great. But in the real Web, with billions of nodes of greatly varying importance, the true probability of being at a node like `www.amazon.com` is orders of magnitude greater than the probability of typical nodes. $\square$

5.1.3 Structure of the Web

It would be nice if the Web were strongly connected like Fig. 5.1. However, it is not, in practice. An early study of the Web found it to have the structure shown in Fig. 5.2. There was a large strongly connected component (SCC), but there were several other portions that were almost as large.

1. The *in-component*, consisting of pages that could reach the SCC by following links, but were not reachable from the SCC.

2. The *out-component*, consisting of pages reachable from the SCC but unable to reach the SCC.

3. *Tendrils*, which are of two types. Some tendrils consist of pages reachable from the in-component but not able to reach the in-component. The other tendrils can reach the out-component, but are not reachable from the out-component.

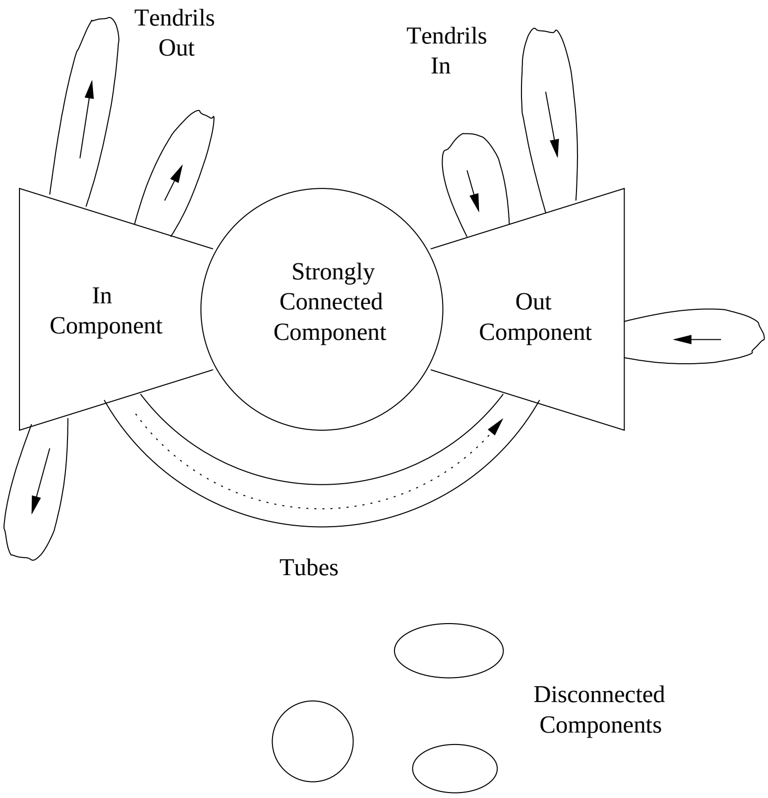

Figure 5.2: The "bowtie" picture of the Web

In addition, there were small numbers of pages found either in

(a) *Tubes*, which are pages reachable from the in-component and able to reach the out-component, but unable to reach the SCC or be reached from the SCC.

(b) Isolated components that are unreachable from the large components (the SCC, in- and out-components) and unable to reach those components.

Several of these structures violate the assumptions needed for the Markov-process iteration to converge to a limit. For example, when a random surfer enters the out-component, they can never leave. As a result, surfers starting in either the SCC or in-component are going to wind up in either the out-component or a tendril off the in-component. Thus, no page in the SCC or in-component winds up with any probability of a surfer being there. If we interpret this probability as measuring the importance of a page, then we conclude falsely that nothing in the SCC or in-component is of any importance.

As a result, PageRank is usually modified to prevent such anomalies. There are really two problems we need to avoid. First is the dead end, a page that has no links out. Surfers reaching such a page disappear, and the result is that in the limit no page that can reach a dead end can have any PageRank at all. The second problem is groups of pages that all have outlinks but they never link to any other pages. These structures are called *spider traps*.[3] Both these problems are solved by a method called "taxation," where we assume a random surfer has a finite probability of leaving the Web at any step, and new surfers are started at each page. We shall illustrate this process as we study each of the two problem cases.

5.1.4 Avoiding Dead Ends

Recall that a page with no link out is called a dead end. If we allow dead ends, the transition matrix of the Web is no longer stochastic, since some of the columns will sum to 0 rather than 1. A matrix whose column sums are at most 1 is called *substochastic*. If we compute $M^i\mathbf{v}$ for increasing powers of a substochastic matrix M, then some or all of the components of the vector go to 0. That is, importance "drains out" of the Web, and we get no information about the relative importance of pages.

Example 5.3 : In Fig. 5.3 we have modified Fig. 5.1 by removing the arc from C to A. Thus, C becomes a dead end. In terms of random surfers, when a surfer reaches C they disappear at the next round. The matrix M that describes Fig. 5.3 is

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

[3]They are so called because the programs that crawl the Web, recording pages and links, are often referred to as "spiders." Once a spider enters a spider trap, it can never leave.

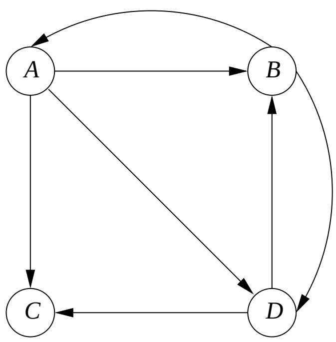

Figure 5.3: C is now a dead end

Note that it is substochastic, but not stochastic, because the sum of the third column, for C, is 0, not 1. Here is the sequence of vectors that result by starting with the vector with each component 1/4, and repeatedly multiplying the vector by M:

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}, \begin{bmatrix} 3/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}, \begin{bmatrix} 5/48 \\ 7/48 \\ 7/48 \\ 7/48 \end{bmatrix}, \begin{bmatrix} 21/288 \\ 31/288 \\ 31/288 \\ 31/288 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

As we see, the probability of a surfer being anywhere goes to 0, as the number of steps increase. □

There are two approaches to dealing with dead ends.

1. We can drop the dead ends from the graph, and also drop their incoming arcs. Doing so may create more dead ends, which also have to be dropped, recursively. However, eventually we wind up with a strongly-connected component, none of whose nodes are dead ends. In terms of Fig. 5.2, recursive deletion of dead ends will remove parts of the out-component, tendrils, and tubes, but leave the SCC and the in-component, as well as parts of any small isolated components.[4]

2. We can modify the process by which random surfers are assumed to move about the Web. This method, which we refer to as "taxation," also solves the problem of spider traps, so we shall defer it to Section 5.1.5.

If we use the first approach, recursive deletion of dead ends, then we solve the remaining graph G by whatever means are appropriate, including the taxation method if there might be spider traps in G. Then, we restore the graph, but keep

[4]You might suppose that the entire out-component and all the tendrils will be removed, but remember that they can have within them smaller strongly connected components, including spider traps, which cannot be deleted.

the PageRank values for the nodes of G. Nodes not in G, but with predecessors all in G can have their PageRank computed by summing, over all predecessors p, the PageRank of p divided by the number of successors of p in the full graph. Now there may be other nodes, not in G, that have the PageRank of all their predecessors computed. These may have their own PageRank computed by the same process. Eventually, all nodes outside G will have their PageRank computed; they can surely be computed in the order opposite to that in which they were deleted.

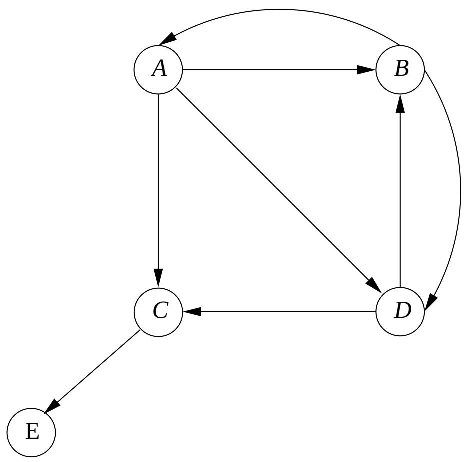

Figure 5.4: A graph with two levels of dead ends

Example 5.4: Figure 5.4 is a variation on Fig. 5.3, where we have introduced a successor E for C. But E is a dead end, and when we remove it, and the arc entering from C, we find that C is now a dead end. After removing C, no more nodes can be removed, since each of A, B, and D have arcs leaving. The resulting graph is shown in Fig. 5.5.

The matrix for the graph of Fig. 5.5 is

$$M = \begin{bmatrix} 0 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 1/2 & 1/2 & 0 \end{bmatrix}$$

The rows and columns correspond to A, B, and D in that order. To get the PageRanks for this matrix, we start with a vector with all components equal to 1/3, and repeatedly multiply by M. The sequence of vectors we get is

$$\begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}, \begin{bmatrix} 1/6 \\ 3/6 \\ 2/6 \end{bmatrix}, \begin{bmatrix} 3/12 \\ 5/12 \\ 4/12 \end{bmatrix}, \begin{bmatrix} 5/24 \\ 11/24 \\ 8/24 \end{bmatrix}, \ldots, \begin{bmatrix} 2/9 \\ 4/9 \\ 3/9 \end{bmatrix}$$

We now know that the PageRank of A is 2/9, the PageRank of B is 4/9, and the PageRank of D is 3/9. We still need to compute PageRanks for C

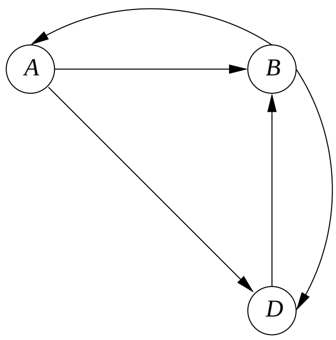

Figure 5.5: The reduced graph with no dead ends

and E, and we do so in the order opposite to that in which they were deleted. Since C was last to be deleted, we know all its predecessors have PageRanks computed. These predecessors are A and D. In Fig. 5.4, A has three successors, so it contributes 1/3 of its PageRank to C. Page D has two successors in Fig. 5.4, so it contributes half its PageRank to C. Thus, the PageRank of C is $\frac{1}{3} \times \frac{2}{9} + \frac{1}{2} \times \frac{3}{9} = 13/54$.

Now we can compute the PageRank for E. That node has only one predecessor, C, and C has only one successor. Thus, the PageRank of E is the same as that of C. Note that the sums of the PageRanks exceed 1, and they no longer represent the distribution of a random surfer. Yet they do represent decent estimates of the relative importance of the pages. □

5.1.5 Spider Traps and Taxation

As we mentioned, a spider trap is a set of nodes with no dead ends but no arcs out. These structures can appear intentionally or unintentionally on the Web, and they cause the PageRank calculation to place all the PageRank within the spider traps.

Example 5.5 : Consider Fig. 5.6, which is Fig. 5.1 with the arc out of C changed to point to C itself. That change makes C a simple spider trap of one node. Note that in general spider traps can have many nodes, and as we shall see in Section 5.4, there are spider traps with millions of nodes that spammers construct intentionally.

The transition matrix for Fig. 5.6 is

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

If we perform the usual iteration to compute the PageRank of the nodes, we

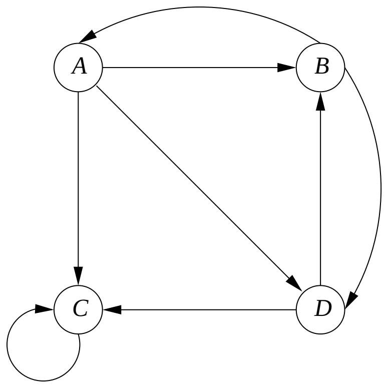

Figure 5.6: A graph with a one-node spider trap

get

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}, \begin{bmatrix} 3/24 \\ 5/24 \\ 11/24 \\ 5/24 \end{bmatrix}, \begin{bmatrix} 5/48 \\ 7/48 \\ 29/48 \\ 7/48 \end{bmatrix}, \begin{bmatrix} 21/288 \\ 31/288 \\ 205/288 \\ 31/288 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

As predicted, all the PageRank is at C, since once there a random surfer can never leave. □

To avoid the problem illustrated by Example 5.5, we modify the calculation of PageRank by allowing each random surfer a small probability of *teleporting* to a random page, rather than following an out-link from their current page. The iterative step, where we compute a new vector estimate of PageRanks $\mathbf{v}'$ from the current PageRank estimate $\mathbf{v}$ and the transition matrix M is

$$\mathbf{v}' = \beta M\mathbf{v} + (1-\beta)\mathbf{e}/n$$

where β is a chosen constant, usually in the range 0.8 to 0.9, $\mathbf{e}$ is a vector of all 1's with the appropriate number of components, and n is the number of nodes in the Web graph. The term $\beta M\mathbf{v}$ represents the case where, with probability β, the random surfer decides to follow an out-link from their present page. The term $(1-\beta)\mathbf{e}/n$ is a vector each of whose components has value $(1-\beta)/n$ and represents the introduction, with probability $1-\beta$, of a new random surfer at a random page.

Note that if the graph has no dead ends, then the probability of introducing a new random surfer is exactly equal to the probability that the random surfer will decide *not* to follow a link from their current page. In this case, it is reasonable to visualize the surfer as deciding either to follow a link or teleport to a random page. However, if there are dead ends, then there is a third possibility, which is that the surfer goes nowhere. Since the term $(1-\beta)\mathbf{e}/n$ does not depend on the sum of the components of the vector $\mathbf{v}$, there will always be some fraction of a surfer operating on the Web. That is, when there are dead ends, the sum of the components of $\mathbf{v}$ may be less than 1, but it will never reach 0.

Example 5.6: Let us see how the new approach to computing PageRank fares on the graph of Fig. 5.6. We shall use $\beta = 0.8$ in this example. Thus, the equation for the iteration becomes

$$\mathbf{v}' = \begin{bmatrix} 0 & 2/5 & 0 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 4/5 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix} \mathbf{v} + \begin{bmatrix} 1/20 \\ 1/20 \\ 1/20 \\ 1/20 \end{bmatrix}$$

Notice that we have incorporated the factor β into M by multiplying each of its elements by $4/5$. The components of the vector $(1-\beta)\mathbf{e}/n$ are each $1/20$, since $1 - \beta = 1/5$ and $n = 4$. Here are the first few iterations:

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}, \begin{bmatrix} 9/60 \\ 13/60 \\ 25/60 \\ 13/60 \end{bmatrix}, \begin{bmatrix} 41/300 \\ 53/300 \\ 153/300 \\ 53/300 \end{bmatrix}, \begin{bmatrix} 543/4500 \\ 707/4500 \\ 2543/4500 \\ 707/4500 \end{bmatrix}, \ldots, \begin{bmatrix} 15/148 \\ 19/148 \\ 95/148 \\ 19/148 \end{bmatrix}$$

By being a spider trap, C has managed to get more than half of the PageRank for itself. However, the effect has been limited, and each of the nodes gets some of the PageRank. □

5.1.6 Using PageRank in a Search Engine

Having seen how to calculate the PageRank vector for the portion of the Web that a search engine has crawled, we should examine how this information is used. Each search engine has a secret formula that decides the order in which to show pages to the user in response to a search query consisting of one or more search terms (words). Google is said to use over 250 different properties of pages, from which a linear order of pages is decided.

First, in order to be considered for the ranking at all, a page has to have at least one of the search terms in the query. Normally, the weighting of properties is such that unless all the search terms are present, a page has very little chance of being in the top ten that are normally shown first to the user. Among the qualified pages, a score is computed for each, and an important component of this score is the PageRank of the page. Other components include the presence or absence of search terms in prominent places, such as headers or the links to the page itself.

5.1.7 Exercises for Section 5.1

Exercise 5.1.1: Compute the PageRank of each page in Fig. 5.7, assuming no taxation.

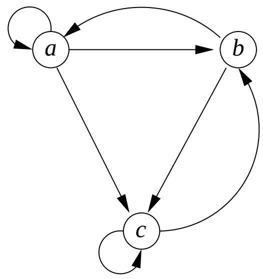

Figure 5.7: An example graph for exercises

Exercise 5.1.2: Compute the PageRank of each page in Fig. 5.7, assuming $\beta = 0.8$.

! **Exercise 5.1.3:** Suppose the Web consists of a *clique* (set of nodes with all possible arcs from one to another) of n nodes and a single additional node that is the successor of each of the n nodes in the clique. Figure 5.8 shows this graph for the case $n = 4$. Determine the PageRank of each page, as a function of n and β.

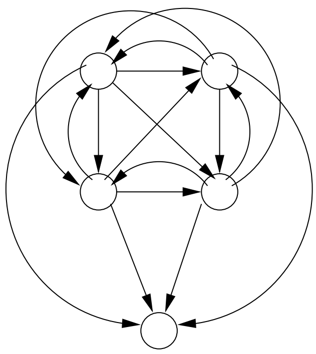

Figure 5.8: Example of graphs discussed in Exercise 5.1.3

!! **Exercise 5.1.4:** Construct, for any integer n, a Web such that, depending on β, any of the n nodes can have the highest PageRank among those n. It is allowed for there to be other nodes in the Web besides these n.

! **Exercise 5.1.5:** Show by induction on n that if the second, third, and fourth components of a vector $\mathbf{v}$ are equal, and M is the transition matrix of Example 5.1, then the second, third, and fourth components are also equal in $M^n\mathbf{v}$ for any $n \geq 0$.

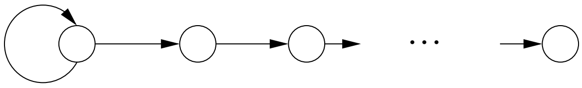

Figure 5.9: A chain of dead ends

Exercise 5.1.6: Suppose we recursively eliminate dead ends from the graph, solve the remaining graph, and estimate the PageRank for the dead-end pages as described in Section 5.1.4. Suppose the graph is a chain of dead ends, headed by a node with a self-loop, as suggested in Fig. 5.9. What would be the PageRank assigned to each of the nodes?

Exercise 5.1.7: Repeat Exercise 5.1.6 for the tree of dead ends suggested by Fig. 5.10. That is, there is a single node with a self-loop, which is also the root of a complete binary tree of n levels.

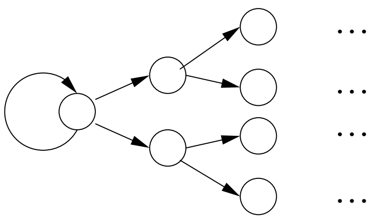

Figure 5.10: A tree of dead ends

5.2 Efficient Computation of PageRank

To compute the PageRank for a large graph representing the Web, we have to perform a matrix–vector multiplication on the order of 50 times, until the vector is close to unchanged at one iteration. To a first approximation, the MapReduce method given in Section 2.3.1 is suitable. However, we must deal with two issues:

1. The transition matrix of the Web M is very sparse. Thus, representing it by all its elements is highly inefficient. Rather, we want to represent the matrix by its nonzero elements.

2. We may not be using MapReduce, or for efficiency reasons we may wish to use a combiner (see Section 2.2.4) with the Map tasks to reduce the amount of data that must be passed from Map tasks to Reduce tasks. In this case, the striping approach discussed in Section 2.3.1 is not sufficient to avoid heavy use of disk (thrashing).

We discuss the solution to these two problems in this section.

5.2.1 Representing Transition Matrices

The transition matrix is very sparse, since the average Web page has about 10 out-links. If, say, we are analyzing a graph of ten billion pages, then only one in a billion entries is not 0. The proper way to represent any sparse matrix is to list the locations of the nonzero entries and their values. If we use 4-byte integers for coordinates of an element and an 8-byte double-precision number for the value, then we need 16 bytes per nonzero entry. That is, the space needed is linear in the number of nonzero entries, rather than quadratic in the side of the matrix.

However, for a transition matrix of the Web, there is one further compression that we can do. If we list the nonzero entries by column, then we know what each nonzero entry is; it is 1 divided by the out-degree of the page. We can thus represent a column by one integer for the out-degree, and one integer per nonzero entry in that column, giving the row number where that entry is located. Thus, we need slightly more than 4 bytes per nonzero entry to represent a transition matrix.

Example 5.7: Let us reprise the example Web graph from Fig. 5.1, whose transition matrix is

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

Recall that the rows and columns represent nodes A, B, C, and D, in that order. In Fig. 5.11 is a compact representation of this matrix.[5]

Source	Degree	Destinations
A	3	B, C, D
B	2	A, D
C	1	A
D	2	B, C

Figure 5.11: Represent a transition matrix by the out-degree of each node and the list of its successors

For instance, the entry for A has degree 3 and a list of three successors. From that row of Fig. 5.11 we can deduce that the column for A in matrix M has 0 in the row for A (since it is not on the list of destinations) and 1/3 in the rows for B, C, and D. We know that the value is 1/3 because the degree column in Fig. 5.11 tells us there are three links out of A. □

[5] Because M is not sparse, this representation is not very useful for M. However, the example illustrates the process of representing matrices in general, and the sparser the matrix is, the more this representation will save.

5.2.2 PageRank Iteration Using MapReduce

One iteration of the PageRank algorithm involves taking an estimated PageRank vector $\mathbf{v}$ and computing the next estimate $\mathbf{v}'$ by

$$\mathbf{v}' = \beta M\mathbf{v} + (1-\beta)\mathbf{e}/n$$

Recall β is a constant slightly less than 1, $\mathbf{e}$ is a vector of all 1's, and n is the number of nodes in the graph that transition matrix M represents.

If n is small enough that each Map task can store the full vector $\mathbf{v}$ in main memory and also have room in main memory for the result vector $\mathbf{v}'$, then there is little more here than a matrix–vector multiplication. The additional steps are to multiply each component of $M\mathbf{v}$ by constant β and to add $(1-\beta)/n$ to each component.

However, it is likely, given the size of the Web today, that $\mathbf{v}$ is much too large to fit in main memory. As we discussed in Section 2.3.1, the method of striping, where we break M into vertical stripes (see Fig. 2.4) and break $\mathbf{v}$ into corresponding horizontal stripes, will allow us to execute the MapReduce process efficiently, with no more of $\mathbf{v}$ at any one Map task than can conveniently fit in main memory.

5.2.3 Use of Combiners to Consolidate the Result Vector

There are two reasons the method of Section 5.2.2 might not be adequate.

1. We might wish to add terms for $\mathbf{v}'_i$, the ith component of the result vector $\mathbf{v}$, at the Map tasks. This improvement is the same as using a combiner, since the Reduce function simply adds terms with a common key. Recall that for a MapReduce implementation of matrix–vector multiplication, the key is the value of i for which a term $m_{ij}\mathbf{v}_j$ is intended.

2. We might not be using MapReduce at all, but rather executing the iteration step at a single machine or a collection of machines.

We shall assume that we are trying to implement a combiner in conjunction with a Map task; the second case uses essentially the same idea.

Suppose that we are using the stripe method to partition a matrix and vector that do not fit in main memory. Then a vertical stripe from the matrix M and a horizontal stripe from the vector $\mathbf{v}$ will contribute to all components of the result vector $\mathbf{v}'$. Since that vector is the same length as $\mathbf{v}$, it will not fit in main memory either. Moreover, as M is stored column-by-column for efficiency reasons, a column can affect any of the components of $\mathbf{v}'$. As a result, it is unlikely that when we need to add a term to some component $\mathbf{v}'_i$, that component will already be in main memory. Thus, most terms will require that a page be brought into main memory to add it to the proper component. That situation, called *thrashing*, takes orders of magnitude too much time to be feasible.

An alternative strategy is based on partitioning the matrix into k^2 blocks, while the vectors are still partitioned into k stripes. A picture, showing the division for $k = 4$, is in Fig. 5.12. Note that we have not shown the multiplication of the matrix by β or the addition of $(1 - \beta)\mathbf{e}/n$, because these steps are straightforward, regardless of the strategy we use.

$$\begin{bmatrix} \mathbf{v}'_1 \\ \mathbf{v}'_2 \\ \mathbf{v}'_3 \\ \mathbf{v}'_4 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \\ \mathbf{v}_4 \end{bmatrix}$$

Figure 5.12: Partitioning a matrix into square blocks

In this method, we use k^2 Map tasks. Each task gets one square of the matrix M, say M_{ij}, and one stripe of the vector $\mathbf{v}$, which must be $\mathbf{v}_j$. Notice that each stripe of the vector is sent to k different Map tasks; $\mathbf{v}_j$ is sent to the task handling M_{ij} for each of the k possible values of i. Thus, $\mathbf{v}$ is transmitted over the network k times. However, each piece of the matrix is sent only once. Since the size of the matrix, properly encoded as described in Section 5.2.1, can be expected to be several times the size of the vector, the transmission cost is not too much greater than the minimum possible. And because we are doing considerable combining at the Map tasks, we save as data is passed from the Map tasks to the Reduce tasks.

The advantage of this approach is that we can keep both the jth stripe of $\mathbf{v}$ and the ith stripe of $\mathbf{v}'$ in main memory as we process M_{ij}. Note that all terms generated from M_{ij} and $\mathbf{v}_j$ contribute to $\mathbf{v}'_i$ and no other stripe of $\mathbf{v}'$.

5.2.4 Representing Blocks of the Transition Matrix

Since we are representing transition matrices in the special way described in Section 5.2.1, we need to consider how the blocks of Fig. 5.12 are represented. Unfortunately, the space required for a column of blocks (a "stripe" as we called it earlier) is greater than the space needed for the stripe as a whole, but not too much greater.

For each block, we need data about all those columns that have at least one nonzero entry within the block. If k, the number of stripes in each dimension, is large, then most columns will have nothing in most blocks of its stripe. For a given block, we not only have to list those rows that have a nonzero entry for that column, but we must repeat the out-degree for the node represented by the column. Consequently, it is possible that the out-degree will be repeated as many times as the out-degree itself. That observation bounds from above the space needed to store the blocks of a stripe at twice the space needed to store the stripe as a whole.

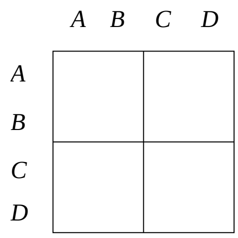

Figure 5.13: A four-node graph is divided into four 2-by-2 blocks

Example 5.8 : Let us suppose the matrix from Example 5.7 is partitioned into blocks, with $k = 2$. That is, the upper-left quadrant represents links from A or B to A or B, the upper-right quadrant represents links from C or D to A or B, and so on. It turns out that in this small example, the only entry that we can avoid is the entry for C in M_{22}, because C has no arcs to either C or D. The tables representing each of the four blocks are shown in Fig. 5.14.

If we examine Fig. 5.14(a), we see the representation of the upper-left quadrant. Notice that the degrees for A and B are the same as in Fig. 5.11, because we need to know the entire number of successors, not the number of successors within the relevant block. However, each successor of A or B is represented in Fig. 5.14(a) or Fig. 5.14(c), but not both. Notice also that in Fig. 5.14(d), there is no entry for C, because there are no successors of C within the lower half of the matrix (rows C and D). □

5.2.5 Other Efficient Approaches to PageRank Iteration

The algorithm discussed in Section 5.2.3 is not the only option. We shall discuss several other approaches that use fewer processors. These algorithms share with the algorithm of Section 5.2.3 the good property that the matrix M is read only once, although the vector $\mathbf{v}$ is read k times, where the parameter k is chosen so that $1/k$th of the vectors $\mathbf{v}$ and $\mathbf{v}'$ can be held in main memory. Recall that the algorithm of Section 5.2.3 uses k^2 processors, assuming all Map tasks are executed in parallel at different processors.

We can assign all the blocks in one row of blocks to a single Map task, and thus reduce the number of Map tasks to k. For instance, in Fig. 5.12, M_{11}, M_{12}, M_{13}, and M_{14} would be assigned to a single Map task. If we represent the blocks as in Fig. 5.14, we can read the blocks in a row of blocks one-at-a-time, so the matrix does not consume a significant amount of main-memory. At the same time that we read M_{ij}, we must read the vector stripe $\mathbf{v}_j$. As a result, each of the k Map tasks reads the entire vector $\mathbf{v}$, along with $1/k$th of the matrix.

Source	Degree	Destinations
A	3	B
B	2	A

(a) Representation of M_{11} connecting A and B to A and B

Source	Degree	Destinations
C	1	A
D	2	B

(b) Representation of M_{12} connecting C and D to A and B

Source	Degree	Destinations
A	3	C, D
B	2	D

(c) Representation of M_{21} connecting A and B to C and D

Source	Degree	Destinations
D	2	C

(d) Representation of M_{22} connecting C and D to C and D

Figure 5.14: Sparse representation of the blocks of a matrix

The work reading M and $\mathbf{v}$ is thus the same as for the algorithm of Section 5.2.3, but the advantage of this approach is that each Map task can combine all the terms for the portion $\mathbf{v}'_i$ for which it is exclusively responsible. In other words, the Reduce tasks have nothing to do but to concatenate the pieces of $\mathbf{v}'$ received from the k Map tasks.

We can extend this idea to an environment in which MapReduce is not used. Suppose we have a single processor, with M and $\mathbf{v}$ stored on its disk, using the same sparse representation for M that we have discussed. We can first simulate the first Map task, the one that uses blocks M_{11} through M_{1k} and all of $\mathbf{v}$ to compute $\mathbf{v}'_1$. Then we simulate the second Map task, reading M_{21} through M_{2k} and all of $\mathbf{v}$ to compute $\mathbf{v}'_2$, and so on. As for the previous algorithms, we thus read M once and $\mathbf{v}$ k times. We can make k as small as possible, subject to the constraint that there is enough main memory to store $1/k$th of $\mathbf{v}$ and $1/k$th of $\mathbf{v}'$, along with as small a portion of M as we can read from disk (typically, one disk block).

5.2.6 Exercises for Section 5.2

Exercise 5.2.1: Suppose we wish to store an $n \times n$ Boolean matrix (0 and 1 elements only). We could represent it by the bits themselves, or we could represent the matrix by listing the positions of the 1's as pairs of integers, each integer requiring $\lceil \log_2 n \rceil$ bits. The former is suitable for dense matrices; the latter is suitable for sparse matrices. How sparse must the matrix be (i.e., what fraction of the elements should be 1's) for the sparse representation to save space?

Exercise 5.2.2: Using the method of Section 5.2.1, represent the transition matrices of the following graphs:

(a) Figure 5.4.

(b) Figure 5.7.

Exercise 5.2.3: Using the method of Section 5.2.4, represent the transition matrices of the graph of Fig. 5.3, assuming blocks have side 2.

Exercise 5.2.4: Consider a Web graph that is a chain, like Fig. 5.9, with n nodes. As a function of k, which you may assume divides n, describe the representation of the transition matrix for this graph, using the method of Section 5.2.4

5.3 Topic-Sensitive PageRank

There are several improvements we can make to PageRank. One, to be studied in this section, is that we can weight certain pages more heavily because of their topic. The mechanism for enforcing this weighting is to alter the way random surfers behave, having them prefer to land on a page that is known to cover the chosen topic. In the next section, we shall see how the topic-sensitive idea can also be applied to negate the effects of a new kind of spam, called "'link spam," that has developed to try to fool the PageRank algorithm.

5.3.1 Motivation for Topic-Sensitive Page Rank

Different people have different interests, and sometimes distinct interests are expressed using the same term in a query. The canonical example is the search query `jaguar`, which might refer to the animal, the automobile, a version of the MAC operating system, or even an ancient game console. If a search engine can deduce that the user is interested in automobiles, for example, then it can do a better job of returning relevant pages to the user.

Ideally, each user would have a private PageRank vector that gives the importance of each page to that user. It is not feasible to store a vector of length many billions for each of a billion users, so we need to do something simpler. The *topic-sensitive PageRank* approach creates one vector for each of some small number of topics, biasing the PageRank to favor pages of that topic. We then endeavour to classify users according to the degree of their interest in each of the selected topics. While we surely lose some accuracy, the benefit is that we store only a short vector for each user, rather than an enormous vector for each user.

Example 5.9: One useful topic set is the 16 top-level categories (sports, medicine, etc.) of the Open Directory (DMOZ).[6] We could create 16 PageRank vectors, one for each topic. If we could determine that the user is interested in one of these topics, perhaps by the content of the pages they have recently viewed, then we could use the PageRank vector for that topic when deciding on the ranking of pages. □

5.3.2 Biased Random Walks

Suppose we have identified some pages that represent a topic such as "sports." To create a topic-sensitive PageRank for sports, we can arrange that the random surfers are introduced only to a random sports page, rather than to a random page of any kind. The consequence of this choice is that random surfers are likely to be at an identified sports page, or a page reachable along a short path from one of these known sports pages. Our intuition is that pages linked to by sports pages are themselves likely to be about sports. The pages they link to are also likely to be about sports, although the probability of being about sports surely decreases as the distance from an identified sports page increases.

The mathematical formulation for the iteration that yields topic-sensitive PageRank is similar to the equation we used for general PageRank. The only difference is how we add the new surfers. Suppose S is a set of integers consisting of the row/column numbers for the pages we have identified as belonging to a certain topic (called the *teleport set*). Let $\mathbf{e}_S$ be a vector that has 1 in the components in S and 0 in other components. Then the *topic-sensitive PageRank for S* is the limit of the iteration

$$\mathbf{v}' = \beta M\mathbf{v} + (1-\beta)\mathbf{e}_S/|S|$$

Here, as usual, M is the transition matrix of the Web, and $|S|$ is the size of set S.

Example 5.10: Let us reconsider the original Web graph we used in Fig. 5.1, which we reproduce as Fig. 5.15. Suppose we use $\beta = 0.8$. Then the transition matrix for this graph, multiplied by β, is

$$\beta M = \begin{bmatrix} 0 & 2/5 & 4/5 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix}$$

[6] This directory, found at `www.dmoz.org`, is a collection of human-classified Web pages.

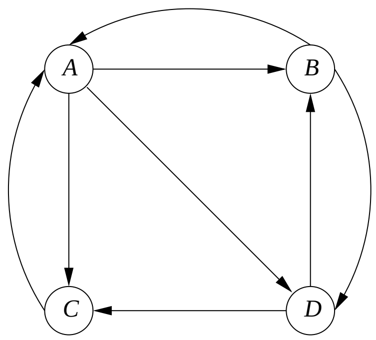

Figure 5.15: Repeat of example Web graph

Suppose that our topic is represented by the teleport set $S = \{B, D\}$. Then the vector $(1 - \beta)\mathbf{e}_S/|S|$ has 1/10 for its second and fourth components and 0 for the other two components. The reason is that $1 - \beta = 1/5$, the size of S is 2, and $\mathbf{e}_S$ has 1 in the components for B and D and 0 in the components for A and C. Thus, the equation that must be iterated is

$$\mathbf{v}' = \begin{bmatrix} 0 & 2/5 & 4/5 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix} \mathbf{v} + \begin{bmatrix} 0 \\ 1/10 \\ 0 \\ 1/10 \end{bmatrix}$$

Here are the first few iterations of this equation. We have also started with the surfers only at the pages in the teleport set. Although the initial distribution has no effect on the limit, it may help the computation to converge faster.

$$\begin{bmatrix} 0/2 \\ 1/2 \\ 0/2 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 2/10 \\ 3/10 \\ 2/10 \\ 3/10 \end{bmatrix}, \begin{bmatrix} 42/150 \\ 41/150 \\ 26/150 \\ 41/150 \end{bmatrix}, \begin{bmatrix} 62/250 \\ 71/250 \\ 46/250 \\ 71/250 \end{bmatrix}, \ldots, \begin{bmatrix} 54/210 \\ 59/210 \\ 38/210 \\ 59/210 \end{bmatrix}$$

Notice that because of the concentration of surfers at B and D, these nodes get a higher PageRank than they did in Example 5.2. In that example, A was the node of highest PageRank. □

5.3.3 Using Topic-Sensitive PageRank

In order to integrate topic-sensitive PageRank into a search engine, we must:

1. Decide on the topics for which we shall create specialized PageRank vectors.

2. Pick a teleport set for each of these topics, and use that set to compute the topic-sensitive PageRank vector for that topic.

3. Find a way of determining the topic or set of topics that are most relevant for a particular search query.

4. Use the PageRank vectors for that topic or topics in the ordering of the responses to the search query.

We have mentioned one way of selecting the topic set: use the top-level topics of the Open Directory. Other approaches are possible, but there is probably a need for human classification of at least some pages.

The third step is probably the trickiest, and several methods have been proposed. Some possibilities:

(a) Allow the user to select a topic from a menu.

(b) Infer the topic(s) by the words that appear in the Web pages recently searched by the user, or recent queries issued by the user. We need to discuss how one goes from a collection of words to a topic, and we shall do so in Section 5.3.4

(c) Infer the topic(s) by information about the user, e.g., their bookmarks or their stated interests on Facebook.

5.3.4 Inferring Topics from Words

The question of classifying documents by topic is a subject that has been studied for decades, and we shall not go into great detail here. Suffice it to say that topics are characterized by words that appear surprisingly often in documents on that topic. For example, neither `fullback` nor `measles` appear very often in documents on the Web. But `fullback` will appear far more often than average in pages about sports, and `measles` will appear far more often than average in pages about medicine.

If we examine the entire Web, or a large, random sample of the Web, we can get the background frequency of each word. Suppose we then go to a large sample of pages known to be about a certain topic, say the pages classified under sports by the Open Directory. Examine the frequencies of words in the sports sample, and identify the words that appear significantly more frequently in the sports sample than in the background. In making this judgment, we must be careful to avoid some extremely rare word that appears in the sports sample with relatively higher frequency. This word is probably a misspelling that happened to appear only in one or a few of the sports pages. Thus, we probably want to put a floor on the number of times a word appears, before it can be considered characteristic of a topic.

Once we have identified a large collection of words that appear much more frequently in the sports sample than in the background, and we do the same for all the topics on our list, we can examine other pages and classify them by topic. Here is a simple approach. Suppose that $S_1, S_2, \ldots, S_k$ are the sets of words that have been determined to be characteristic of each of the topics on our list. Let P be the set of words that appear in a given page P. Compute the Jaccard similarity (recall Section 3.1.1) between P and each of the S_i's. Classify the page as that topic with the highest Jaccard similarity. Note that all Jaccard similarities may be very low, especially if the sizes of the sets S_i are small. Thus, it is important to pick reasonably large sets S_i to make sure that we cover all aspects of the topic represented by the set.

We can use this method, or a number of variants, to classify the pages the user has most recently retrieved. We could say the user is interested in the topic into which the largest number of these pages fall. Or we could blend the topic-sensitive PageRank vectors in proportion to the fraction of these pages that fall into each topic, thus constructing a single PageRank vector that reflects the user's current blend of interests. We could also use the same procedure on the pages that the user currently has bookmarked, or combine the bookmarked pages with the recently viewed pages.

5.3.5 Exercises for Section 5.3

Exercise 5.3.1: Compute the topic-sensitive PageRank for the graph of Fig. 5.15, assuming the teleport set is:

(a) A only.

(b) A and C.

5.4 Link Spam

When it became apparent that PageRank and other techniques used by Google made term spam ineffective, spammers turned to methods designed to fool the PageRank algorithm into overvaluing certain pages. The techniques for artificially increasing the PageRank of a page are collectively called *link spam*. In this section we shall first examine how spammers create link spam, and then see several methods for decreasing the effectiveness of these spamming techniques, including TrustRank and measurement of spam mass.

5.4.1 Architecture of a Spam Farm

A collection of pages whose purpose is to increase the PageRank of a certain page or pages is called a *spam farm*. Figure 5.16 shows the simplest form of spam farm. From the point of view of the spammer, the Web is divided into three parts:

1. *Inaccessible pages*: the pages that the spammer cannot affect. Most of the Web is in this part.

2. *Accessible pages*: those pages that, while they are not controlled by the spammer, can be affected by the spammer.

3. *Own pages*: the pages that the spammer owns and controls.

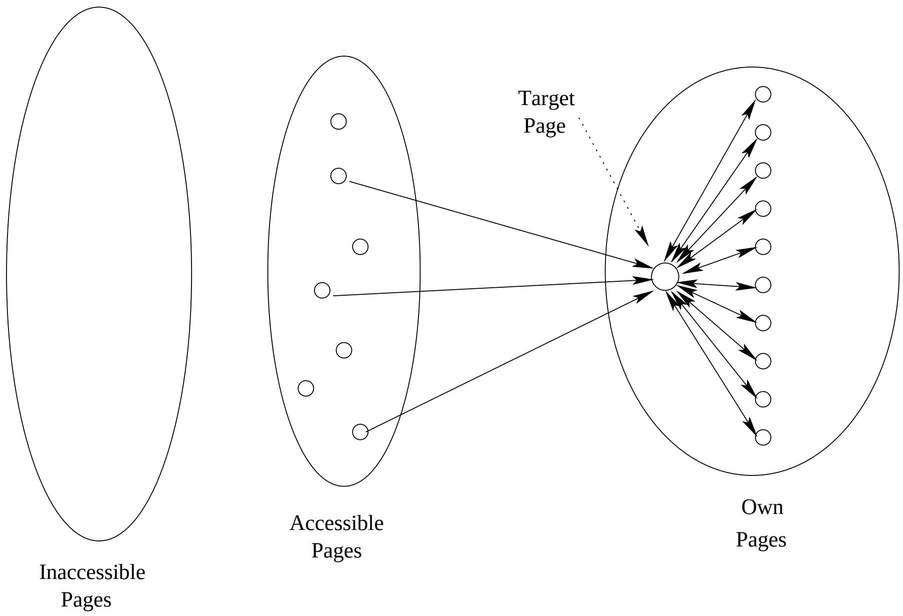

Figure 5.16: The Web from the point of view of the link spammer

The spam farm consists of the spammer's own pages, organized in a special way as seen on the right, and some links from the accessible pages to the spammer's pages. Without some links from the outside, the spam farm would be useless, since it would not even be crawled by a typical search engine.

Concerning the accessible pages, it might seem surprising that one can affect a page without owning it. However, today there are many sites, such as blogs or newspapers that invite others to post their comments on the site. In order to get as much PageRank flowing to his own pages from outside, the spammer posts many comments such as "I agree. Please see my article at www.mySpamFarm.com."

In the spam farm, there is one page t, the *target page*, at which the spammer attempts to place as much PageRank as possible. There are a large number m of *supporting* pages, that accumulate the portion of the PageRank that is distributed equally to all pages (the fraction $1 - \beta$ of the PageRank that represents surfers going to a random page). The supporting pages also prevent the PageRank of t from being lost, to the extent possible, since some will be taxed away at each round. Notice that t has a link to every supporting page, and every supporting page links only to t.

5.4.2 Analysis of a Spam Farm

Suppose that PageRank is computed using a taxation parameter β, typically around 0.85. That is, β is the fraction of a page's PageRank that gets distributed to its successors at the next round. Let there be n pages on the Web in total, and let some of them be a spam farm of the form suggested in Fig. 5.16, with a target page t and m supporting pages. Let x be the amount of PageRank contributed by the accessible pages. That is, x is the sum, over all accessible pages p with a link to t, of the PageRank of p times β, divided by the number of successors of p. Finally, let y be the unknown PageRank of t. We shall solve for y.

First, the PageRank of each supporting page is

$$\beta y/m + (1-\beta)/n$$

The first term represents the contribution from t. The PageRank y of t is taxed, so only βy is distributed to t's successors. That PageRank is divided equally among the m supporting pages. The second term is the supporting page's share of the fraction $1-\beta$ of the PageRank that is divided equally among all pages on the Web.

Now, let us compute the PageRank y of target page t. Its PageRank comes from three sources:

1. Contribution x from outside, as we have assumed.

2. β times the PageRank of every supporting page; that is,

$$\beta\big(\beta y/m + (1-\beta)/n\big)$$

3. $(1-\beta)/n$, the share of the fraction $1-\beta$ of the PageRank that belongs to t. This amount is negligible and will be dropped to simplify the analysis.

Thus, from (1) and (2) above, we can write

$$y = x + \beta m\Big(\frac{\beta y}{m} + \frac{1-\beta}{n}\Big) = x + \beta^2 y + \beta(1-\beta)\frac{m}{n}$$

We may solve the above equation for y, yielding

$$y = \frac{x}{1-\beta^2} + c\frac{m}{n}$$

where $c = \beta(1-\beta)/(1-\beta^2) = \beta/(1+\beta)$.

Example 5.11: If we choose $\beta = 0.85$, then $1/(1-\beta^2) = 3.6$, and $c = \beta/(1+\beta) = 0.46$. That is, the structure has amplified the external PageRank contribution by 360%, and also obtained an amount of PageRank that is 46% of the fraction of the Web, m/n, that is in the spam farm. □

5.4.3 Combating Link Spam

It has become essential for search engines to detect and eliminate link spam, just as it was necessary in the previous decade to eliminate term spam. There are two approaches to link spam. One is to look for structures such as the spam farm in Fig. 5.16, where one page links to a very large number of pages, each of which links back to it. Search engines surely search for such structures and eliminate those pages from their index. That causes spammers to develop different structures that have essentially the same effect of capturing PageRank for a target page or pages. There is essentially no end to variations of Fig. 5.16, so this war between the spammers and the search engines will likely go on for a long time.

However, there is another approach to eliminating link spam that doesn't rely on locating the spam farms. Rather, a search engine can modify its definition of PageRank to lower the rank of link-spam pages automatically. We shall consider two different formulas:

1. *TrustRank*, a variation of topic-sensitive PageRank designed to lower the score of spam pages.

2. *Spam mass*, a calculation that identifies the pages that are likely to be spam and allows the search engine to eliminate those pages or to lower their PageRank strongly.

5.4.4 TrustRank

TrustRank is topic-sensitive PageRank, where the "topic" is a set of pages believed to be trustworthy (not spam). The theory is that while a spam page might easily be made to link to a trustworthy page, it is unlikely that a trustworthy page would link to a spam page. The borderline area is a site with blogs or other opportunities for spammers to create links, as was discussed in Section 5.4.1. These pages cannot be considered trustworthy, even if their own content is highly reliable, as would be the case for a reputable newspaper that allowed readers to post comments.

To implement TrustRank, we need to develop a suitable teleport set of trustworthy pages. Two approaches that have been tried are:

1. Let humans examine a set of pages and decide which of them are trustworthy. For example, we might pick the pages of highest PageRank to examine, on the theory that, while link spam can raise a page's rank from the bottom to the middle of the pack, it is essentially impossible to give a spam page a PageRank near the top of the list.

2. Pick a domain whose membership is controlled, on the assumption that it is hard for a spammer to get their pages into these domains. For example, we could pick the .edu domain, since university pages are unlikely to be spam farms. We could likewise pick .mil, or .gov. However, the problem

with these specific choices is that they are almost exclusively US sites. To get a good distribution of trustworthy Web pages, we should include the analogous sites from foreign countries, e.g., ac.il, or edu.sg.

It is likely that search engines today implement a strategy of the second type routinely, so that what we think of as PageRank really is a form of TrustRank.

5.4.5 Spam Mass

The idea behind spam mass is that we measure for each page the fraction of its PageRank that comes from spam. We do so by computing both the ordinary PageRank and the TrustRank based on some teleport set of trustworthy pages. Suppose page p has PageRank r and TrustRank t. Then the *spam mass* of p is $(r-t)/r$. A negative or small positive spam mass means that p is probably not a spam page, while a spam mass close to 1 suggests that the page probably is spam. It is possible to eliminate pages with a high spam mass from the index of Web pages used by a search engine, thus eliminating a great deal of the link spam without having to identify particular structures that spam farmers use.

Example 5.12 : Let us consider both the PageRank and topic-sensitive PageRank that were computed for the graph of Fig. 5.1 in Examples 5.2 and 5.10, respectively. In the latter case, the teleport set was nodes B and D, so let us assume those are the trusted pages. Figure 5.17 tabulates the PageRank, TrustRank, and spam mass for each of the four nodes.

Node	PageRank	TrustRank	Spam Mass
A	3/9	54/210	0.229
B	2/9	59/210	-0.264
C	2/9	38/210	0.186
D	2/9	59/210	-0.264

Figure 5.17: Calculation of spam mass

In this simple example, the only conclusion is that the nodes B and D, which were a priori determined not to be spam, have negative spam mass and are therefore not spam. The other two nodes, A and C, each have a positive spam mass, since their PageRanks are higher than their TrustRanks. For instance, the spam mass of A is computed by taking the difference $3/9 - 54/210 = 8/105$ and dividing $8/105$ by the PageRank $3/9$ to get $8/35$ or about 0.229. However, their spam mass is still closer to 0 than to 1, so it is probable that they are not spam. □

5.4.6 Exercises for Section 5.4

Exercise 5.4.1 : In Section 5.4.2 we analyzed the spam farm of Fig. 5.16, where every supporting page links back to the target page. Repeat the analysis for a spam farm in which:

(a) Each supporting page links to itself instead of to the target page.

(b) Each supporting page links nowhere.

(c) Each supporting page links both to itself and to the target page.

Exercise 5.4.2: For the original Web graph of Fig. 5.1, assuming only B is a trusted page:

(a) Compute the TrustRank of each page.

(b) Compute the spam mass of each page.

! **Exercise 5.4.3:** Suppose two spam farmers agree to link their spam farms. How would you link the pages in order to increase as much as possible the PageRank of each spam farm's target page? Is there an advantage to linking spam farms?

5.5 Hubs and Authorities

An idea called "hubs and authorities' was proposed shortly after PageRank was first implemented. The algorithm for computing hubs and authorities bears some resemblance to the computation of PageRank, since it also deals with the iterative computation of a fixedpoint involving repeated matrix–vector multiplication. However, there are also significant differences between the two ideas, and neither can substitute for the other.

This hubs-and-authorities algorithm, sometimes called *HITS* (*hyperlink-induced topic search*), was originally intended not as a preprocessing step before handling search queries, as PageRank is, but as a step to be done along with the processing of a search query, to rank only the responses to that query. We shall, however, describe it as a technique for analyzing the entire Web, or the portion crawled by a search engine. There is reason to believe that something like this approach is, in fact, used by the Ask search engine.

5.5.1 The Intuition Behind HITS

While PageRank assumes a one-dimensional notion of importance for pages, HITS views important pages as having two flavors of importance.

1. Certain pages are valuable because they provide information about a topic. These pages are called *authorities*.

2. Other pages are valuable not because they provide information about any topic, but because they tell you where to go to find out about that topic. These pages are called *hubs*.

Example 5.13 : A typical department at a university maintains a Web page listing all the courses offered by the department, with links to a page for each course, telling about the course – the instructor, the text, an outline of the course content, and so on. If you want to know about a certain course, you need the page for that course; the departmental course list will not do. The course page is an authority for that course. However, if you want to find out what courses the department is offering, it is not helpful to search for each courses' page; you need the page with the course list first. This page is a hub for information about courses. □

Just as PageRank uses the recursive definition of importance that "a page is important if important pages link to it," HITS uses a mutually recursive definition of two concepts: "a page is a good hub if it links to good authorities, and a page is a good authority if it is linked to by good hubs."

5.5.2 Formalizing Hubbiness and Authority

To formalize the above intuition, we shall assign two scores to each Web page. One score represents the *hubbiness* of a page – that is, the degree to which it is a good hub, and the second score represents the degree to which the page is a good authority. Assuming that pages are enumerated, we represent these scores by vectors $\mathbf{h}$ and $\mathbf{a}$. The ith component of $\mathbf{h}$ gives the hubbiness of the ith page, and the ith component of $\mathbf{a}$ gives the authority of the same page.

While importance is divided among the successors of a page, as expressed by the transition matrix of the Web, the normal way to describe the computation of hubbiness and authority is to add the authority of successors to estimate hubbiness and to add hubbiness of predecessors to estimate authority. If that is all we did, then the hubbiness and authority values would typically grow beyond bounds. Thus, we normally scale the values of the vectors $\mathbf{h}$ and $\mathbf{a}$ so that the largest component is 1. An alternative is to scale so that the sum of components is 1.

To describe the iterative computation of $\mathbf{h}$ and $\mathbf{a}$ formally, we use the *link matrix of the Web*, L. If we have n pages, then L is an $n \times n$ matrix, and $L_{ij} = 1$ if there is a link from page i to page j, and $L_{ij} = 0$ if not. We shall also have need for L^{T}, the *transpose* of L. That is, $L^{\mathrm{T}}_{ij} = 1$ if there is a link from page j to page i, and $L^{\mathrm{T}}_{ij} = 0$ otherwise. Notice that L^{T} is similar to the matrix M that we used for PageRank, but where L^{T} has 1, M has a fraction – 1 divided by the number of out-links from the page represented by that column.

Example 5.14 : For a running example, we shall use the Web of Fig. 5.4, which we reproduce here as Fig. 5.18. An important observation is that dead ends or spider traps do not prevent the HITS iteration from converging to a meaningful pair of vectors. Thus, we can work with Fig. 5.18 directly, with no "taxation" or alteration of the graph needed. The link matrix L and its transpose are shown in Fig. 5.19. □

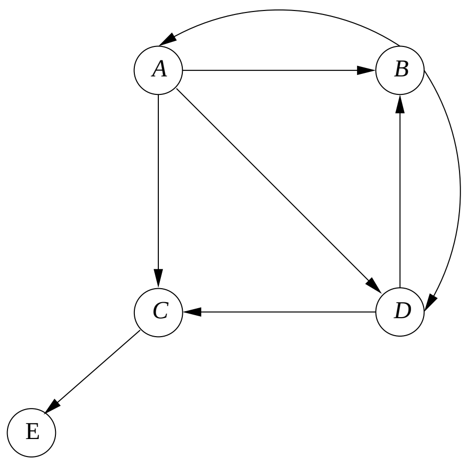

Figure 5.18: Sample data used for HITS examples

$$L = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad L^{\mathrm{T}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 5.19: The link matrix for the Web of Fig. 5.18 and its transpose

The fact that the hubbiness of a page is proportional to the sum of the authority of its successors is expressed by the equation $\mathbf{h} = \lambda L\mathbf{a}$, where λ is an unknown constant representing the scaling factor needed. Likewise, the fact that the authority of a page is proportional to the sum of the hubbinesses of its predecessors is expressed by $\mathbf{a} = \mu L^{\mathrm{T}}\mathbf{h}$, where μ is another scaling constant. These equations allow us to compute the hubbiness and authority independently, by substituting one equation in the other, as:

- $\mathbf{h} = \lambda\mu LL^{\mathrm{T}}\mathbf{h}$.
- $\mathbf{a} = \lambda\mu L^{\mathrm{T}}L\mathbf{a}$.

However, since LL^{T} and $L^{\mathrm{T}}L$ are not as sparse as L and L^{T}, we are usually better off computing $\mathbf{h}$ and $\mathbf{a}$ in a true mutual recursion. That is, start with $\mathbf{h}$ a vector of all 1's.

1. Compute $\mathbf{a} = L^{\mathrm{T}}\mathbf{h}$ and then scale so the largest component is 1.
2. Next, compute $\mathbf{h} = L\mathbf{a}$ and scale again.

Now, we have a new **h** and can repeat steps (1) and (2) until at some iteration the changes to the two vectors are sufficiently small that we can stop and accept the current values as the limit.

$$\underset{\mathbf{h}}{\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}} \quad \underset{L^{\mathrm{T}}\mathbf{h}}{\begin{bmatrix} 1 \\ 2 \\ 2 \\ 2 \\ 1 \end{bmatrix}} \quad \underset{\mathbf{a}}{\begin{bmatrix} 1/2 \\ 1 \\ 1 \\ 1 \\ 1/2 \end{bmatrix}} \quad \underset{L\mathbf{a}}{\begin{bmatrix} 3 \\ 3/2 \\ 1/2 \\ 2 \\ 0 \end{bmatrix}} \quad \underset{\mathbf{h}}{\begin{bmatrix} 1 \\ 1/2 \\ 1/6 \\ 2/3 \\ 0 \end{bmatrix}}$$

$$\underset{L^{\mathrm{T}}\mathbf{h}}{\begin{bmatrix} 1/2 \\ 5/3 \\ 5/3 \\ 3/2 \\ 1/6 \end{bmatrix}} \quad \underset{\mathbf{a}}{\begin{bmatrix} 3/10 \\ 1 \\ 1 \\ 9/10 \\ 1/10 \end{bmatrix}} \quad \underset{L\mathbf{a}}{\begin{bmatrix} 29/10 \\ 6/5 \\ 1/10 \\ 2 \\ 0 \end{bmatrix}} \quad \underset{\mathbf{h}}{\begin{bmatrix} 1 \\ 12/29 \\ 1/29 \\ 20/29 \\ 0 \end{bmatrix}}$$

Figure 5.20: First two iterations of the HITS algorithm

Example 5.15 : Let us perform the first two iterations of the HITS algorithm on the Web of Fig. 5.18. In Fig. 5.20 we see the succession of vectors computed. The first column is the initial **h**, all 1's. In the second column, we have estimated the relative authority of pages by computing $L^{\mathrm{T}}\mathbf{h}$, thus giving each page the sum of the hubbinesses of its predecessors. The third column gives us the first estimate of **a**. It is computed by scaling the second column; in this case we have divided each component by 2, since that is the largest value in the second column.

The fourth column is $L\mathbf{a}$. That is, we have estimated the hubbiness of each page by summing the estimate of the authorities of each of its successors. Then, the fifth column scales the fourth column. In this case, we divide by 3, since that is the largest value in the fourth column. Columns six through nine repeat the process outlined in our explanations for columns two through five, but with the better estimate of hubbiness given by the fifth column.

The limit of this process may not be obvious, but it can be computed by a simple program. The limits are:

$$\mathbf{h} = \begin{bmatrix} 1 \\ 0.3583 \\ 0 \\ 0.7165 \\ 0 \end{bmatrix} \qquad \mathbf{a} = \begin{bmatrix} 0.2087 \\ 1 \\ 1 \\ 0.7913 \\ 0 \end{bmatrix}$$

This result makes sense. First, we notice that the hubbiness of E is surely 0, since it leads nowhere. The hubbiness of C depends only on the authority of E and vice versa, so it should not surprise us that both are 0. A is the greatest hub, since it links to the three biggest authorities, B, C, and D. Also, B and C are the greatest authorities, since they are linked to by the two biggest hubs, A and D.

For Web-sized graphs, the only way of computing the solution to the hubs-and-authorities equations is iteratively. However, for this tiny example, we can compute the solution by solving equations. We shall use the equations $\mathbf{h} = \lambda\mu LL^{\mathrm{T}}\mathbf{h}$. First, LL^{T} is

$$LL^{\mathrm{T}} = \begin{bmatrix} 3 & 1 & 0 & 2 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Let $\nu = 1/(\lambda\mu)$ and let the components of $\mathbf{h}$ for nodes A through E be a through e, respectively. Then the equations for $\mathbf{h}$ can be written

$$\begin{array}{ll} \nu a = 3a + b + 2d & \nu b = a + 2b \\ \nu c = c & \nu d = 2a + 2d \\ \nu e = 0 & \end{array}$$

The equation for b tells us $b = a/(\nu - 2)$ and the equation for d tells us $d = 2a/(\nu - 2)$. If we substitute these expressions for b and d in the equation for a, we get $\nu a = a\bigl(3+5/(\nu-2)\bigr)$. From this equation, since a is a factor of both sides, we are left with a quadratic equation for ν which simplifies to $\nu^2 - 5\nu + 1 = 0$. The positive root is $\nu = (5 + \sqrt{21})/2 = 4.791$. Now that we know ν is neither 0 or 1, the equations for c and e tell us immediately that $c = e = 0$.

Finally, if we recognize that a is the largest component of $\mathbf{h}$ and set $a = 1$, we get $b = 0.3583$ and $d = 0.7165$. Along with $c = e = 0$, these values give us the limiting value of h. The value of $\mathbf{a}$ can be computed from $\mathbf{h}$ by multiplying by L^{T} and scaling. □

5.5.3 Exercises for Section 5.5

Exercise 5.5.1: Compute the hubbiness and authority of each of the nodes in our original Web graph of Fig. 5.1.

! **Exercise 5.5.2:** Suppose our graph is a chain of n nodes, as was suggested by Fig. 5.9. Compute the hubs and authorities vectors, as a function of n.

5.6 Summary of Chapter 5

✦ *Term Spam*: Early search engines were unable to deliver relevant results because they were vulnerable to term spam – the introduction into Web pages of words that misrepresented what the page was about.

- ✦ *The Google Solution to Term Spam*: Google was able to counteract term spam by two techniques. First was the PageRank algorithm for determining the relative importance of pages on the Web. The second was a strategy of believing what other pages said about a given page, in or near their links to that page, rather than believing only what the page said about itself.

- ✦ *PageRank*: PageRank is an algorithm that assigns a real number, called its PageRank, to each page on the Web. The PageRank of a page is a measure of how important the page is, or how likely it is to be a good response to a search query. In its simplest form, PageRank is a solution to the recursive equation "a page is important if important pages link to it."

- ✦ *Transition Matrix of the Web*: We represent links in the Web by a matrix whose ith row and ith column represent the ith page of the Web. If there are one or more links from page j to page i, then the entry in row i and column j is $1/k$, where k is the number of pages to which page j links. Other entries of the transition matrix are 0.

- ✦ *Computing PageRank on Strongly Connected Web Graphs*: For strongly connected Web graphs (those where any node can reach any other node), PageRank is the principal eigenvector of the transition matrix. We can compute PageRank by starting with any nonzero vector and repeatedly multiplying the current vector by the transition matrix, to get a better estimate.[7] After about 50 iterations, the estimate will be very close to the limit, which is the true PageRank.

- ✦ *The Random Surfer Model*: Calculation of PageRank can be thought of as simulating the behavior of many random surfers, who each start at a random page and at any step move, at random, to one of the pages to which their current page links. The limiting probability of a surfer being at a given page is the PageRank of that page. The intuition is that people tend to create links to the pages they think are useful, so random surfers will tend to be at a useful page.

- ✦ *Dead Ends*: A dead end is a Web page with no links out. The presence of dead ends will cause the PageRank of some or all of the pages to go to 0 in the iterative computation, including pages that are not dead ends. We can eliminate all dead ends before undertaking a PageRank calculation by recursively dropping nodes with no arcs out. Note that dropping one node can cause another, which linked only to it, to become a dead end, so the process must be recursive.

[7] Technically, the condition for this method to work is more restricted than simply "strongly connected." However, the other necessary conditions will surely be met by any large strongly connected component of the Web that was not artificially constructed.

✦ *Spider Traps*: A spider trap is a set of nodes that, while they may link to each other, have no links out to other nodes. In an iterative calculation of PageRank, the presence of spider traps cause all the PageRank to be captured within that set of nodes.

✦ *Taxation Schemes*: To counter the effect of spider traps (and of dead ends, if we do not eliminate them), PageRank is normally computed in a way that modifies the simple iterative multiplication by the transition matrix. A parameter β is chosen, typically around 0.85. Given an estimate of the PageRank, the next estimate is computed by multiplying the estimate by β times the transition matrix, and then adding $(1-\beta)/n$ to the estimate for each page, where n is the total number of pages.

✦ *Taxation and Random Surfers*: The calculation of PageRank using taxation parameter β can be thought of as giving each random surfer a probability $1-\beta$ of leaving the Web, and introducing an equivalent number of surfers randomly throughout the Web.

✦ *Efficient Representation of Transition Matrices*: Since a transition matrix is very sparse (almost all entries are 0), it saves both time and space to represent it by listing its nonzero entries. However, in addition to being sparse, the nonzero entries have a special property: they are all the same in any given column; the value of each nonzero entry is the inverse of the number of nonzero entries in that column. Thus, the preferred representation is column-by-column, where the representation of a column is the number of nonzero entries, followed by a list of the rows where those entries occur.

✦ *Very Large-Scale Matrix–Vector Multiplication*: For Web-sized graphs, it may not be feasible to store the entire PageRank estimate vector in the main memory of one machine. Thus, we can break the vector into k segments and break the transition matrix into k^2 squares, called blocks, assigning each square to one machine. The vector segments are each sent to k machines, so there is a small additional cost in replicating the vector.

✦ *Representing Blocks of a Transition Matrix*: When we divide a transition matrix into square blocks, the columns are divided into k segments. To represent a segment of a column, nothing is needed if there are no nonzero entries in that segment. However, if there are one or more nonzero entries, then we need to represent the segment of the column by the total number of nonzero entries in the column (so we can tell what value the nonzero entries have) followed by a list of the rows with nonzero entries.

✦ *Topic-Sensitive PageRank*: If we know the queryer is interested in a certain topic, then it makes sense to bias the PageRank in favor of pages on that topic. To compute this form of PageRank, we identify a set of pages known to be on that topic, and we use it as a "teleport set." The

PageRank calculation is modified so that only the pages in the teleport set are given a share of the tax, rather than distributing the tax among all pages on the Web.

✦ *Creating Teleport Sets*: For topic-sensitive PageRank to work, we need to identify pages that are very likely to be about a given topic. One approach is to start with the pages that the open directory (DMOZ) identifies with that topic. Another is to identify words known to be associated with the topic, and select for the teleport set those pages that have an unusually high number of occurrences of such words.

✦ *Link Spam*: To fool the PageRank algorithm, unscrupulous actors have created spam farms. These are collections of pages whose purpose is to concentrate high PageRank on a particular target page.

✦ *Structure of a Spam Farm*: Typically, a spam farm consists of a target page and very many supporting pages. The target page links to all the supporting pages, and the supporting pages link only to the target page. In addition, it is essential that some links from outside the spam farm be created. For example, the spammer might introduce links to their target page by writing comments in other people's blogs or discussion groups.

✦ *TrustRank*: One way to ameliorate the effect of link spam is to compute a topic-sensitive PageRank called TrustRank, where the teleport set is a collection of trusted pages. For example, the home pages of universities could serve as the trusted set. This technique avoids sharing the tax in the PageRank calculation with the large numbers of supporting pages in spam farms and thus preferentially reduces their PageRank.

✦ *Spam Mass*: To identify spam farms, we can compute both the conventional PageRank and the TrustRank for all pages. Those pages that have much lower TrustRank than PageRank are likely to be part of a spam farm.

✦ *Hubs and Authorities*: While PageRank gives a one-dimensional view of the importance of pages, an algorithm called HITS tries to measure two different aspects of importance. Authorities are those pages that contain valuable information. Hubs are pages that, while they do not themselves contain the information, link to places where the information can be found.

✦ *Recursive Formulation of the HITS Algorithm*: Calculation of the hubs and authorities scores for pages depends on solving the recursive equations: "a hub links to many authorities, and an authority is linked to by many hubs." The solution to these equations is essentially an iterated matrix–vector multiplication, just like PageRank's. However, the existence of dead ends or spider traps does not affect the solution to the

HITS equations in the way they do for PageRank, so no taxation scheme is necessary.

5.7 References for Chapter 5

The PageRank algorithm was first expressed in [1]. The experiments on the structure of the Web, which we used to justify the existence of dead ends and spider traps, were described in [2]. The block-stripe method for performing the PageRank iteration is taken from [5].

Topic-sensitive PageRank is taken from [6]. TrustRank is described in [4], and the idea of spam mass is taken from [3].

The HITS (hubs and authorities) idea was described in [7].

1. S. Brin and L. Page, "Anatomy of a large-scale hypertextual web search engine," *Proc. 7th Intl. World-Wide-Web Conference*, pp. 107–117, 1998.

2. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Weiner, "Graph structure in the web," *Computer Networks* **33**:1–6, pp. 309–320, 2000.

3. Z. Gyöngi, P. Berkhin, H. Garcia-Molina, and J. Pedersen, "Link spam detection based on mass estimation," *Proc. 32nd Intl. Conf. on Very Large Databases*, pp. 439–450, 2006.

4. Z. Gyöngi, H. Garcia-Molina, and J. Pedersen, "Combating link spam with trustrank," *Proc. 30th Intl. Conf. on Very Large Databases*, pp. 576–587, 2004.

5. T.H. Haveliwala, "Efficient computation of PageRank," Stanford Univ. Dept. of Computer Science technical report, Sept., 1999. Available as

   `http://infolab.stanford.edu/~taherh/papers/efficient-pr.pdf`

6. T.H. Haveliwala, "Topic-sensitive PageRank," *Proc. 11th Intl. World-Wide-Web Conference*, pp. 517–526, 2002

7. J.M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM* **46**:5, pp. 604–632, 1999.