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.

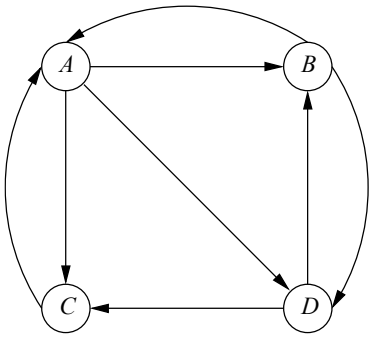

Figure 5.1: A hypothetical example of the Web

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. □

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.

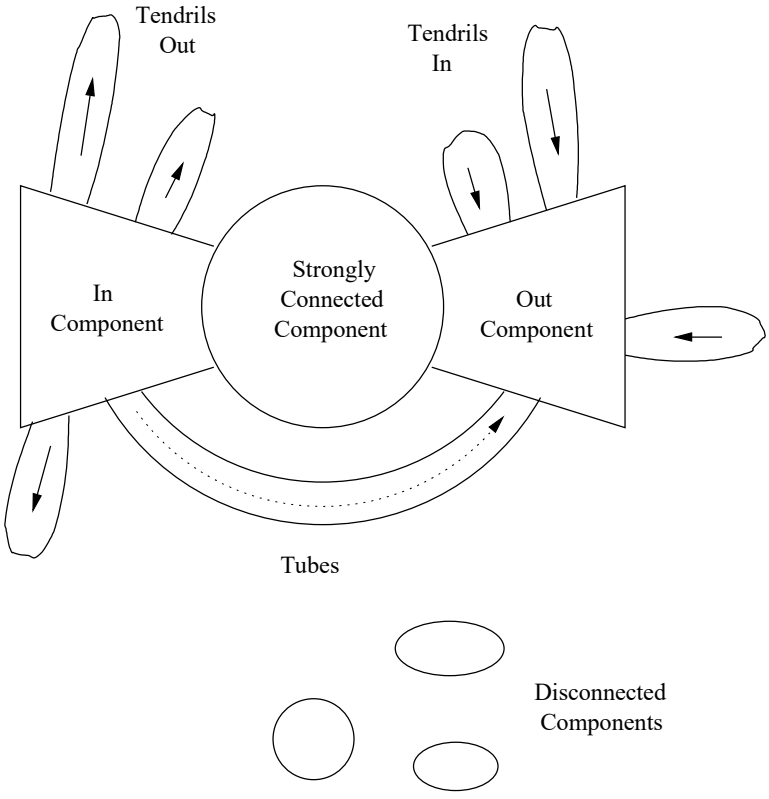

Figure 5.2: The "bowtie" picture of the Web

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.

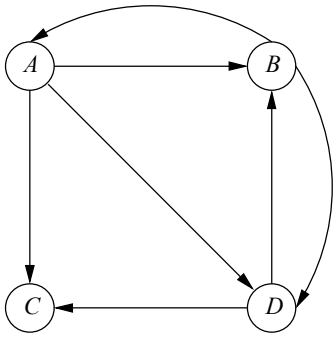

Figure 5.3: C is now a dead end

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:

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

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.

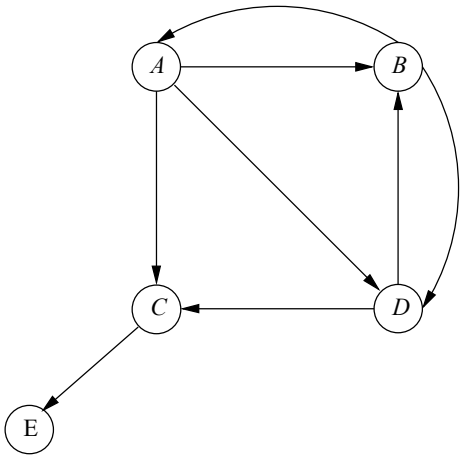

Figure 5.4: A graph with two levels of dead ends

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

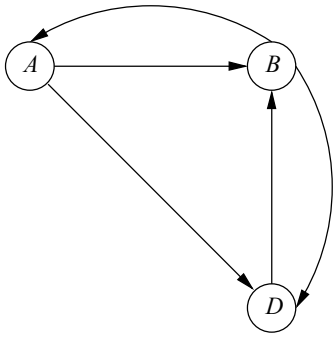

Figure 5.5: The reduced graph with no dead ends

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

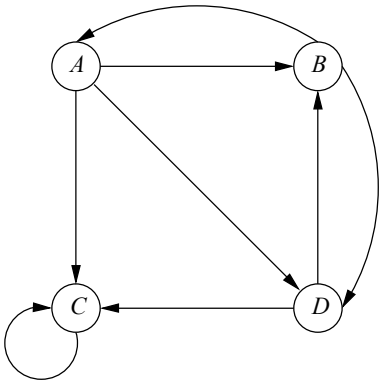

Figure 5.6: A graph with a one-node spider trap

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.

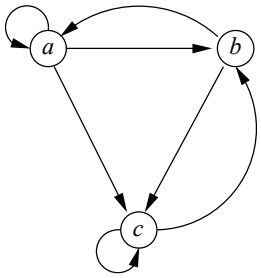

Figure 5.7: An example graph for exercises

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.

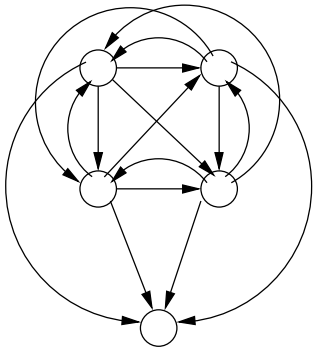

Figure 5.8: Example of graphs discussed in Exercise 5.1.3

!! **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 $n \geq 0$.

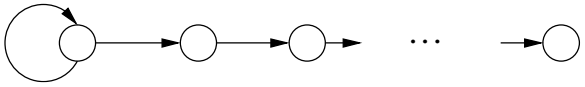

Figure 5.9: A chain of dead ends

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.

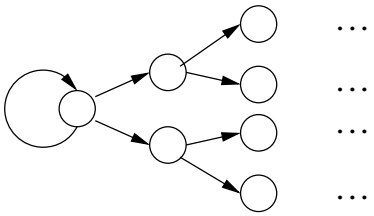

Figure 5.10: A tree of dead ends

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.

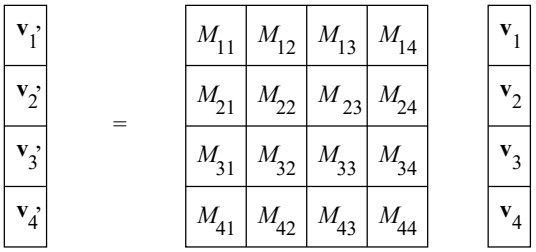

Figure 5.12: Partitioning a matrix into square blocks

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.

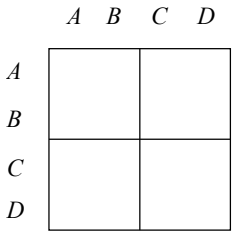

Figure 5.13: A four-node graph is divided into four 2-by-2 blocks

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