5.3 PageRank

PageRank [117] is a measure of web page quality based on the structure of the hyperlink graph. Although it is only one of thousands of features that is taken into account in Google's search algorithm, it is perhaps one of the best known and most studied.

A vivid way to illustrate PageRank is to imagine a random web surfer: the surfer visits a page, randomly clicks a link on that page, and repeats ad infinitum. PageRank is a measure of how frequently a page would be encountered

by our tireless web surfer. More precisely, PageRank is a probability distribution over nodes in the graph representing the likelihood that a random walk over the link structure will arrive at a particular node. Nodes that have high in-degrees tend to have high PageRank values, as well as nodes that are linked to by other nodes with high PageRank values. This behavior makes intuitive sense: if PageRank is a measure of page quality, we would expect high-quality pages to contain "endorsements" from many other pages in the form of hyperlinks. Similarly, if a high-quality page links to another page, then the second page is likely to be high quality also. PageRank represents one particular approach to inferring the quality of a web page based on hyperlink structure; two other popular algorithms, not covered here, are SALSA [88] and HITS [84] (also known as "hubs and authorities").

The complete formulation of PageRank includes an additional component. As it turns out, our web surfer doesn't just randomly click links. Before the surfer decides where to go next, a biased coin is flipped—heads, the surfer clicks on a random link on the page as usual. Tails, however, the surfer ignores the links on the page and randomly "jumps" or "teleports" to a completely different page.

But enough about random web surfing. Formally, the PageRank P of a page n is defined as follows:

$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)} \tag{5.1}$$

where $|G|$ is the total number of nodes (pages) in the graph, α is the random jump factor, $L(n)$ is the set of pages that link to n, and $C(m)$ is the out-degree of node m (the number of links on page m). The random jump factor α is sometimes called the "teleportation" factor; alternatively, $(1 - \alpha)$ is referred to as the "damping" factor.

Let us break down each component of the formula in detail. First, note that PageRank is defined recursively—this gives rise to an iterative algorithm we will detail in a bit. A web page n receives PageRank "contributions" from all pages that link to it, $L(n)$. Let us consider a page m from the set of pages $L(n)$: a random surfer at m will arrive at n with probability $1/C(m)$ since a link is selected at random from all outgoing links. Since the PageRank value of m is the probability that the random surfer will be at m, the probability of arriving at n from m is $P(m)/C(m)$. To compute the PageRank of n, we need to sum contributions from all pages that link to n. This is the summation in the second half of the equation. However, we also need to take into account the random jump: there is a $1/|G|$ chance of landing at any particular page, where $|G|$ is the number of nodes in the graph. Of course, the two contributions need to be combined: with probability α the random surfer executes a random jump, and with probability $1 - \alpha$ the random surfer follows a hyperlink.

Note that PageRank assumes a community of honest users who are not trying to "game" the measure. This is, of course, not true in the real world, where an adversarial relationship exists between search engine companies and

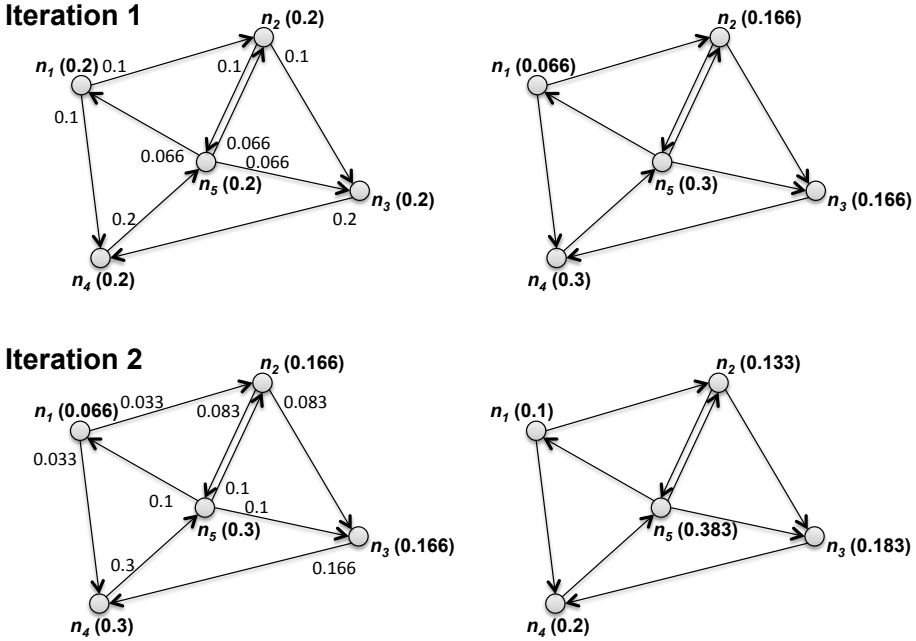

Figure 5.5: PageRank toy example showing two iterations, top and bottom. Left graphs show PageRank values at the beginning of each iteration and how much PageRank mass is passed to each neighbor. Right graphs show updated PageRank values at the end of each iteration.

a host of other organizations and individuals (marketers, spammers, activists, etc.) who are trying to manipulate search results—to promote a cause, product, or service, or in some cases, to trap and intentionally deceive users (see, for example, [12, 63]). A simple example is a so-called "spider trap", a infinite chain of pages (e.g., generated by CGI) that all link to a single page (thereby artificially inflating its PageRank). For this reason, PageRank is only one of thousands of features used in ranking web pages.

The fact that PageRank is recursively defined translates into an iterative algorithm which is quite similar in basic structure to parallel breadth-first search. We start by presenting an informal sketch. At the beginning of each iteration, a node passes its PageRank contributions to other nodes that it is connected to. Since PageRank is a probability distribution, we can think of this as spreading probability mass to neighbors via outgoing links. To conclude the iteration, each node sums up all PageRank contributions that have been passed to it and computes an updated PageRank score. We can think of this as gathering probability mass passed to a node via its incoming links. This algorithm iterates until PageRank values don't change anymore.

Figure 5.5 shows a toy example that illustrates two iterations of the algorithm. As a simplification, we ignore the random jump factor for now (i.e., $\alpha = 0$) and further assume that there are no dangling nodes (i.e., nodes with no outgoing edges). The algorithm begins by initializing a uniform distribution of PageRank values across nodes. In the beginning of the first iteration (top, left), partial PageRank contributions are sent from each node to its neighbors connected via outgoing links. For example, n_1 sends 0.1 PageRank mass to n_2 and 0.1 PageRank mass to n_4. This makes sense in terms of the random surfer model: if the surfer is at n_1 with a probability of 0.2, then the surfer could end up either in n_2 or n_4 with a probability of 0.1 each. The same occurs for all the other nodes in the graph: note that n_5 must split its PageRank mass three ways, since it has three neighbors, and n_4 receives all the mass belonging to n_3 because n_3 isn't connected to any other node. The end of the first iteration is shown in the top right: each node sums up PageRank contributions from its neighbors. Note that since n_1 has only one incoming link, from n_3, its updated PageRank value is smaller than before, i.e., it "passed along" more PageRank mass than it received. The exact same process repeats, and the second iteration in our toy example is illustrated by the bottom two graphs. At the beginning of each iteration, the PageRank values of all nodes sum to one. PageRank mass is preserved by the algorithm, guaranteeing that we continue to have a valid probability distribution at the end of each iteration.

Pseudo-code of the MapReduce PageRank algorithm is shown in Algorithm 5.3; it is simplified in that we continue to ignore the random jump factor and assume no dangling nodes (complications that we will return to later). An illustration of the running algorithm is shown in Figure 5.6 for the first iteration of the toy graph in Figure 5.5. The algorithm maps over the nodes, and for each node computes how much PageRank mass needs to be distributed to its neighbors (i.e., nodes on the adjacency list). Each piece of the PageRank mass is emitted as the value, keyed by the node ids of the neighbors. Conceptually, we can think of this as passing PageRank mass along outgoing edges.

In the shuffle and sort phase, the MapReduce execution framework groups values (piece of PageRank mass) passed along the graph edges by destination node (i.e., all edges that point to the same node). In the reducer, PageRank mass contributions from all incoming edges are summed to arrive at the updated PageRank value for each node. As with the parallel breadth-first search algorithm, the graph structure itself must be passed from iteration to iteration. Each node data structure is emitted in the mapper and written back out to disk in the reducer. All PageRank mass emitted by the mappers are accounted for in the reducer: since we begin with the sum of PageRank values across all nodes equal to one, the sum of all the updated PageRank values should remain a valid probability distribution.

Having discussed the simplified PageRank algorithm in MapReduce, let us now take into account the random jump factor and dangling nodes: as it turns out both are treated similarly. Dangling nodes are nodes in the graph that have no outgoing edges, i.e., their adjacency lists are empty. In the hyperlink graph of the web, these might correspond to pages in a crawl that have not

Algorithm 5.3 PageRank (simplified)

In the map phase we evenly divide up each node's PageRank mass and pass each piece along outgoing edges to neighbors. In the reduce phase PageRank contributions are summed up at each destination node. Each MapReduce job corresponds to one iteration of the algorithm. This algorithm does not handle dangling nodes and the random jump factor.

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sum incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

been downloaded yet. If we simply run the algorithm in Algorithm 5.3 on graphs with dangling nodes, the total PageRank mass will not be conserved, since no key-value pairs will be emitted when a dangling node is encountered in the mappers.

The proper treatment of PageRank mass "lost" at the dangling nodes is to redistribute it across all nodes in the graph evenly (cf. [22]). There are many ways to determine the missing PageRank mass. One simple approach is by instrumenting the algorithm in Algorithm 5.3 with counters: whenever the mapper processes a node with an empty adjacency list, it keeps track of the node's PageRank value in the counter. At the end of the iteration, we can access the counter to find out how much PageRank mass was lost at the dangling nodes.[8] Another approach is to reserve a special key for storing PageRank mass from dangling nodes. When the mapper encounters a dangling node, its PageRank mass is emitted with the special key; the reducer must be modified to contain special logic for handling the missing PageRank mass. Yet another approach is to write out the missing PageRank mass as "side data" for each map task (using the in-mapper combining technique for aggregation); a final pass in the driver program is needed to sum the mass across all map tasks.

[8] In Hadoop, counters are 8-byte integers: a simple workaround is to multiply PageRank values by a large constant, and then cast as an integer.

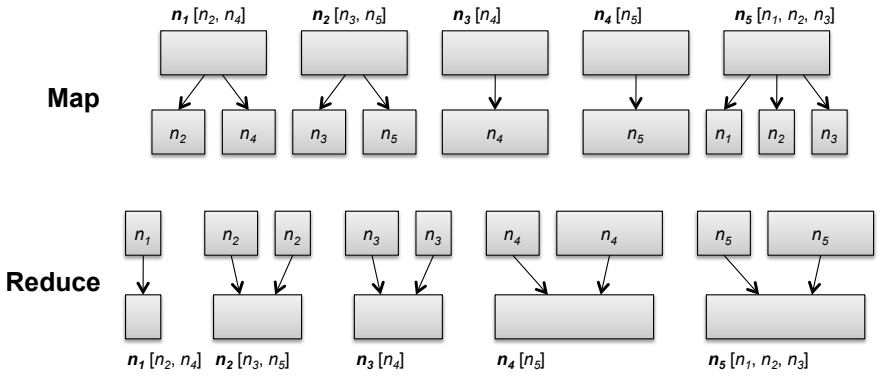

Figure 5.6: Illustration of the MapReduce PageRank algorithm corresponding to the first iteration in Figure 5.5. The size of each box is proportion to its PageRank value. During the map phase, PageRank mass is distributed evenly to nodes on each node's adjacency list (shown at the very top). Intermediate values are keyed by node (shown inside the boxes). In the reduce phase, all partial PageRank contributions are summed together to arrive at updated values.

Either way, we arrive at the amount of PageRank mass lost at the dangling nodes—this then must be redistribute evenly across all nodes.

This redistribution process can be accomplished by mapping over all nodes again. At the same time, we can take into account the random jump factor. For each node, its current PageRank value p is updated to the final PageRank value p' according to the following formula:

$$p' = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \left(\frac{m}{|G|} + p \right) \tag{5.2}$$

where m is the missing PageRank mass, and $|G|$ is the number of nodes in the entire graph. We add the PageRank mass from link traversal (p, computed from before) to the share of the lost PageRank mass that is distributed to each node ($m/|G|$). Finally, we take into account the random jump factor: with probability α the random surfer arrives via jumping, and with probability $1 - \alpha$ the random surfer arrives via incoming links. Note that this MapReduce job requires no reducers.

Putting everything together, one iteration of PageRank requires two MapReduce jobs: the first to distribute PageRank mass along graph edges, and the second to take care of dangling nodes and the random jump factor. At end of each iteration, we end up with exactly the same data structure as the beginning, which is a requirement for the iterative algorithm to work. Also, the PageRank values of all nodes sum up to one, which ensures a valid probability distribution.

Typically, PageRank is iterated until convergence, i.e., when the PageRank values of nodes no longer change (within some tolerance, to take into account, for example, floating point precision errors). Therefore, at the end of each iteration, the PageRank driver program must check to see if convergence has been reached. Alternative stopping criteria include running a fixed number of iterations (useful if one wishes to bound algorithm running time) or stopping when the *ranks* of PageRank values no longer change. The latter is useful for some applications that only care about comparing the PageRank of two arbitrary pages and do not need the actual PageRank values. Rank stability is obtained faster than the actual convergence of values.

In absolute terms, how many iterations are necessary for PageRank to converge? This is a difficult question to *precisely* answer since it depends on many factors, but generally, fewer than one might expect. In the original PageRank paper [117], convergence on a graph with 322 million edges was reached in 52 iterations (see also Bianchini et al. [22] for additional discussion). On today's web, the answer is not very meaningful due to the adversarial nature of web search as previously discussed—the web is full of spam and populated with sites that are actively trying to "game" PageRank and related hyperlink-based metrics. As a result, running PageRank in its unmodified form presented here would yield unexpected and undesirable results. Of course, strategies developed by web search companies to combat link spam are proprietary (and closely-guarded secrets, for obvious reasons)—but undoubtedly these algorithmic modifications impact convergence behavior. A full discussion of the escalating "arms race" between search engine companies and those that seek to promote their sites is beyond the scope of this book.[9]

[9]For the interested reader, the proceedings of a workshop series on Adversarial Information Retrieval (AIRWeb) provide great starting points into the literature.