CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 7, 4/20/2015. Scribed by Olivier Debaillon-Vesque, Arthus de Saint Chaffray, Lan Nguyen, Keenon Werling.

7 Page Rank

We are given as input a directed graph $G = (V, E)$ representing a network of websites. The goal is to find a good metric for measuring the importance of each node in the graph (corresponding to a ranking over the websites). In academia, a paper's importance can be roughly measured by how many citations it receives, and we can use an analogous approach with the web. A website's importance will be measured by the number of sites that link to it. Ideally, we would like the amount of importance conferred on a website by receiving a link to be proportional to the importance of the website giving the link.

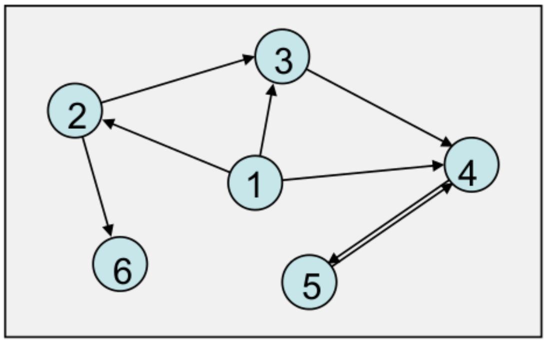

Figure 1: Example network with nodes representing websites and edges representing links

We can formalize this intuition via the process of a random walk. We would like to model a "random-surfer" who is traversing the web with uniform probability of following any link outgoing from the page the surfer is currently on. We are interested in the behavior of this random surfer in the limit as she takes an infinite number of jumps.
To express this in linear algebra, we will make a use of the adjacency matrix, A, and a out-degree matrix D, where:

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$$D = \begin{pmatrix} deg(v_1) & 0 & \dots & 0 \\ 0 & deg(v_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & deg(v_n) \end{pmatrix}$$

Let $Q = D^{-1}A$. This forms the transition matrix of the random-walker. Given the current state of the walker each row of the matrix gives the probability the walker will transition to each new state.

$$Q_{i,j} = \begin{cases} 1/deg(v_i) & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

It's interesting to note that Q^k_{ij} is equal to the probability of going from node i to node j in exactly k steps, in a random walk over graph G.

The stationary distribution of the Markov Chain with the transition probabilities defined by Q is the solution to our PageRank problem as defined above. The stationary distribution specifies what proportion of time on average is spent at specific node during an infinitely long random walk.

One issue with this approach is the fact that chain can 'get stuck' at the nodes that have no outgoing links, and the nodes with no incoming links are never visited in the random walk. This problem can be fixed by giving our random surfer some "teleportation" probability. Intuitively, the random surfer chooses either to follow a link on the current page or to type a page URL into their browser at random. This amounts to adding a matrix Λ with all positive entries to Q. Naively, Λ could consist of all ones, but by changing the weights in Λ we can encode user preferences (perhaps they're more likely to randomly jump to Reddit than a Machine Learning blog, or vice versa).

This means we now will use the following matrix to parameterize our Markov chain describing our random walker.

$$P = \alpha\Lambda + (1 - \alpha)Q$$

where $0 < \alpha < 1$ is the teleport probability, and Λ is a rank 1 matrix whose rows correspond to the distribution of where a teleporting surfer arrives.

We are looking for a row vector π of node probabilities such that:

$$\pi P = \pi$$

We normally solve problems like this via power iteration. That algorithm is very simple. We initialize $V^{(0)}$ to any initial distribution -for example to the uniform vector $V^{(0)} = [1/n, \dots, 1/n]$. Then we follow a converging recurrence:

$$V^{(k+1)} = V^{(k)}P = V^{(0)}P^{k+1}$$

If $\lim_{k\to\infty} P^k = P^\infty$ exists, then the previous recurrence converges to the steady-state distribution. However, each recurrence could take a very long time, since V is a very large vector (the number of sites on the internet), and P is a *very* large matrix (the number of sites on the internet *squared*). We will address that more in a moment.

There is a very convenient theorem here that says that this will converge for tractably small k, as a result of the random jump probabilities.

$$\|V^{(k)} - \pi\|_2 \leq e^{-ak}$$

Where $a \approx 2$. This means that with every iteration, we gain roughly another decimal point of precision. At a very high level, this is possible because adding the random jump probabilities (Λ) acts as a strong regularizer on our Markov chain and causes it to mix very quickly. So for $k \geq 8$, we can guarantee very accurate convergence (to around 8 decimal places). This will make PageRank a tractable algorithm, even though every iteration will require us to perform a web-scale matrix-vector multiply.
Note: we do not normalize the vectors at each iteration because we only need the rank ordering in PageRank and normalizing would add complexity. If we were to carry out a large number of iterations the norm of our computed vectors would grow to infinity, however because we restrict ourselves to 8 iterations, this is not a concern.

7.1 Implementation and Optimization

We next discuss the implementation of PageRank in Spark. The following notes should be read with the corresponding lecture slides.

1. **Assumptions made**:
   - The data is stored in two RDDs, one for links (the matrix Q) and one for rankings (the vector V);
   - Both RDDs are too large to fit in memory on a single machine;
   - Q is a row sparse matrix, so each row fits in memory on a single machine.
   We can make these assumptions because each row corresponds to outgoing edges of a web page.

2. **Naive implementation** (*slides 4 and 5*)
   At each iteration we need to put the current rankings 'next to' the relevant links so that each web page can give out some of its rank to its outgoing edges. This requires a join.
   In the naive implementation we join the links and ranks RDDs every iteration. We then use reduceByKey to sum up all the contributions.
   With this approach, each join and reduceByKey require a full shuffle over the network. These steps are expensive (all-to-all communications).

3. **Pre-partitioning** (*slides 6 and 7*)
   We would like to reduce the number of expensive all-to-all communication steps that are needed.
   We pre-partition the links RDD (the biggest one) so that links for URLs with the same hash code are on the same node. We choose to pre-partition the biggest RDD (links) as each iteration will still require sending one of the RDDs over the network and it is preferable to send the smaller one.

This pre-partitioning requires an all-to-all communication. But, it avoids future all-to-all communications of the links RDD in subsequent joins with the ranks RDD at each iteration. We now need one all-to-all communication per iteration (for the reduceByKey operation), compared to two in the previous naive case. The second and subsequent joins no longer require all-to-all communication since reduceByKey knows where to send the distributions. We therefore half the number all-to-all communications required. We see from slide 11 that using this controlled partitioning produces a significant improvement in the time per iteration.

4. **Remark** (*slide 13*):
   We can do even better: most outgoing links are to web pages with the same domain name. If we were to partition these web pages on the same machine this would reduce communication costs. We use this information to define a custom partitioning: hash URLs names by domain name. Therefore, were possible, all pages with same domain name will appear on the same machine.