CME 323: Distributed Algorithms and Optimization, Spring 2020
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 15, 5/19/2020. Scribed by Robin Brown.

15 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.

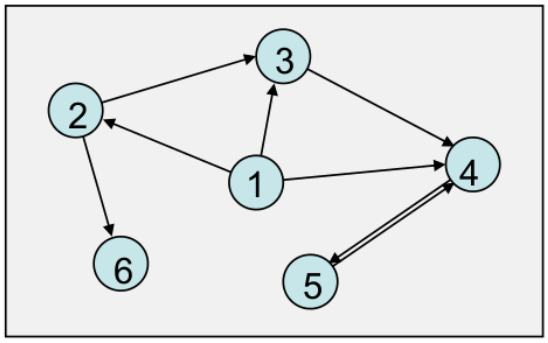

Figure 1: Example network with nodes representing websites and edges representing links

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.

15.1 Partitioning

The motivation for partitioning arises from the fact that sending data via the network is expensive. The hope is that smartly partitioning an RDD in its constituent machines will reduce the need for communication. We highlight the need for partitioning by highlighting an example through the *PageRank* algorithm.

15.1.1 Sparse Pagerank

The problem is to compute node importance given

- RDD of sparse graph `links` which stores the pair inscribing directed edges.
- RDD storing the rank of the nodes. In this situation, we consider guess for node importance which we refine through iterations.

An application is computing importance of urls (nodes) in the internet(graph). One way is to compute the topmost eigenvalue of the adjacency graph matrix A. Let x_t be the guess of the rank. One way to compute this:

$$x_{t+1} \leftarrow Ax_t.$$

Here, it is assumed that entries of the adjacency matrix are 1 if there is an outgoing link and this is normalized by the number of neighbors the node has. We use the following modification:

$$x_{t+1} \leftarrow 0.15\mathbf{1} + 0.85Ax_t.$$

In other words,

1. Each page (node) is started with rank 1. Or x_t is initialized with the uniform vector.

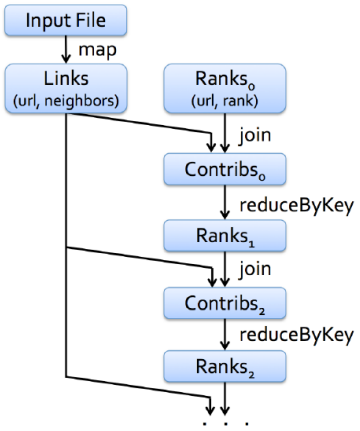

Figure 2: Pagerank without partitioning incurs multiple all-to-all communication from repeated joins.

2. On each iteration page p computes its contribution $(x_t)_p/|\text{neighbors}_p|$ and sends it to its neighbors.

3. Each page's rank is $0.15 + 0.85 \times$ contribs. We are implementing this matrix-vector multiplication using `join` and aligning the RDDs.

The algorithm is now written:

```
val links =  // RDD of (url, neighbors) pairs
var ranks =  RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
        val contribs = links. join(ranks). flatMap {
                case (url,(links,rank)) => links.map(dest => (dest,rank/links.size))
        }
        ranks = contribs. reduceByKey(_+_). mapValues(.15+0.85*_)
}
```

The parts in red are local operations and the parts in blue are distributed operations. *All-to-all* communication is required in the `join` and `reduceByKey` operations. As can be seen from Fig. 2, joins necessitating all-to-all communication are required for each iteration.

We *partition* the links such that all links within the same partition sit on the same machine. This can be done with the following line-

```
val links = sc.textfile(...)....partitionBy(new HashPartitioner(8))
```

Any shuffle operation (such as one with join) will respect the partitioner if set. As can be seen from Fig. 3, new ranks after the first iteration sit in the same machine as the old links and repeated joins do not occur.

A note of caution here is that `map` may change key values and partitioning is not respected. If knowledge of partitioning is to be preserved, `mapValue` can be employed.

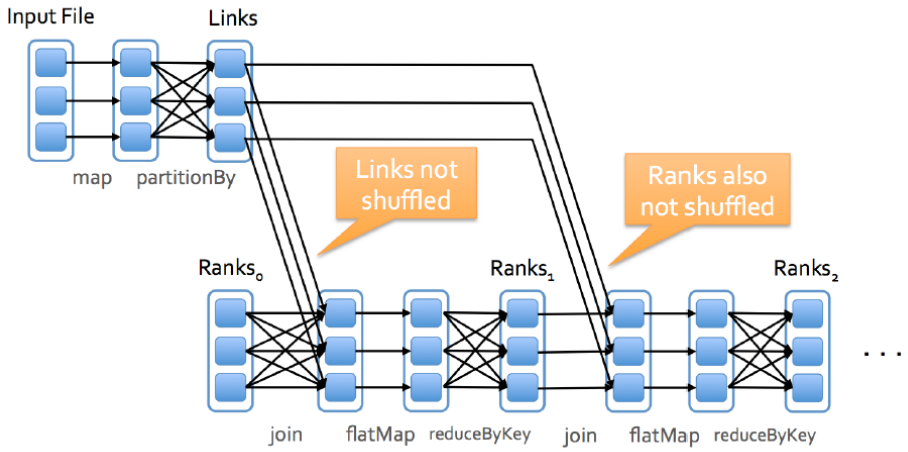

Figure 3: With partitioning, repeated joins does not happen.

Further communication can be cut down from domain knowledge. For example, we know that most links from a webpage are to another page with the same domain name. Thus we can partition all webpages with the same domain in the same machine.

References