Lab 8: PageRank

May 21st, 2014

Objectives

PageRank is one of the key ingredients that made the Google search engine so successful. It is a powerful ranking method that is used in many applications, beyond web search. In this lab, you will implement the PageRank algorithm and get hands-on experience with it. In particular, you will look at

- two different ways of implementing the algorithm,
- pitfalls of naive implementations and how to avoid them,
- and how to improve the PageRank score of a page.

At the end of the lab, you will hopefully have gained some intuition about the algorithm and the meaning of a PageRank score.

Deliverables

The following deliverables, to be handed in before the start of the next lab, should be bundled in a single `zip` or `pdf` file.

- Method `compute` of the class `NaiveRandomSurfer`.
- Method `compute` of the class `RandomSurfer`.
- Method `googleMatrix` of the class `PowerMethod`.
- The first 20 pages with the highest PageRank scores in the Wikipedia graph.
- Method `addIncomingEdges` of the class `Manipulation`.
- Method `addEdges` of the class `Manipulation`.
- The PageRank score of page *History of mathematics* based on your own implementation of the second scenario of improving the PageRank score.

1 The Dataset

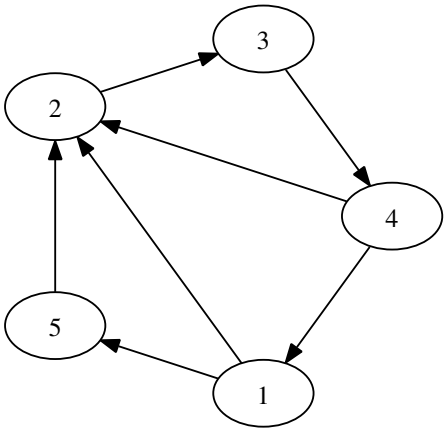

Figure 1: The graph represented by `data/simple.graph`.

We represent the Web as a graph where webpages (nodes) are connected through hyperlinks (directed edges.) In the `data` folder we provide you with four different "webs" on which you will run PageRank. Three of them are simple, artificial toy examples (`simple.graph`, `components.graph`, `absorbing.graph`.) The last one, `wikipedia.graph`, has been extracted from a small version of Wikipedia[1]. Each graph is stored as an adjacency list. For example, `simple.graph`:

```
first       1 4
second      2
third       3
fourth      0 1
fifth       1
```

Figure 1 shows the graphical representation of this file. You can view the other ones[2] using a Python script that is provided in the archive:

```
scripts/show.py path/to/graph
```

2 Random Surfer Model

In this section we will code our first implementation of PageRank, closely following the *Random Surfer* model. In this model, a "surfer" starts by choosing a webpage uniformly at random from the list of all pages; then, she selects a hyperlink on the current page uniformly at random and continues to the next webpage. The surfer continues this process of selecting links at random from successive webpages.

Each time a page is visited, we increase this page's counter. The PageRank score of the webpage is then defined as the normalized number of times that webpage is visited during this random surfing, i.e. it represents the probability that a random surfer happens to be on this page at any moment in time.

- Look at the graph depicted in Figure 1. Intuitively, can you order the 5 nodes by decreasing PageRank score? Imagine that you follow a random walk in the graph and count the proportion of times you land on each node.

[1] *Wikipedia for Schools*, see: `http://schools-wikipedia.org/`.
[2] You might have some trouble visualizing `wikipedia.graph`, because it is quite large.

- Now open Eclipse and start by looking at the file `NaiveRandomSurfer.java`. Complete the method `compute()`. Follow the random surfer idea, take a long random walk and count the number of times you encounter each edge. Normalize the counts by the length of the walk (to get a probability distribution.) You have your first simple PageRank algorithm ready. Test it with the provided unit tests, and run it on `simple.graph`.
- Consider the two graphs `components.graph` and `absorbing.graph`. As explained before, you can use `scripts/draw.py` to draw these graphs. Can you identify any problems if we want to run our naive random surfer implementation on them?
- Run your implementation of `NaiveRandomSurfer` on these two graphs. What happens?

To overcome these issues, two ideas were proposed by the inventors of PageRank (Larry Page and Sergey Brin.)

1. When we reach a dangling node (i.e. a node with no outgoing link), the surfer starts from a new node chosen uniformly at random.
2. At each iteration, with small probability (called the *damping factor*), we start surfing from a new node chosen uniformly at random. This is called a *random restart*. We will use a damping factor of 0.15, as defined in `PageRankAlgorithm`.

You will now implement these ideas in `RandomSurfer.java`.

- Complete the method `compute()` in `RandomSurfer.java` and integrate the two ideas discussed above. Be careful on how to proceed with dangling nodes.
- Test your code with the unit tests, and run it again on the two graphs (`components.graph` and `absorbing.graph`). Do the PageRank results make intuitive sense?

3 Power Iteration Method

The problem with our algorithm so far is that it is very slow to converge and needs a huge number of iterations for big graphs. To overcome this limitation we will use the *power iteration method*, which we explain in this section.

Let $W(V, E)$ be the web graph, with $N = |V|$ nodes, and let o_u be the outdegree of node u (i.e. the number of outgoing links.) The transition matrix of a random walk on the web graph, H, can be written as

$$H_{u,v} = \begin{cases} 1/o_u & (u,v) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

If our random surfer arrives at a dangling node, we want it to continue surfing from a new webpage chosen uniformly at random. We define w as the indicator vector of dangling nodes. The new transition matrix is then defined as

$$\widehat{H} = H + \frac{1}{N}(w\mathbf{1}^T).$$

Graphs with absorbing nodes are not the only classes of graphs which need special care. For example, assume we have a graph with two connected components. A random surfer can walk only in one of the two components: the one where the surfing started. To solve this, we use random restarts. At every iteration, flip a coin, with probability θ walk on $\hat{H}$ and with probability $1 - \theta$ jump to a new random webpage. The final transition matrix G, called the *Google matrix*, is thus defined as

$$G = \theta\widehat{H} + (1-\theta)\frac{\mathbf{1}\mathbf{1}^T}{N}. \tag{1}$$

Finding the PageRank is then equivalent to find the (unique) distribution π that satisfies

$$\pi^T = \pi^T G.$$

One way to solve this equation[3] is the power iteration method. Given an initial vector $\pi^{(0)}$ the power iteration method successively computes

$$\pi^{(k)} = \pi^{(k-1)} G, k = 1, 2, \ldots, \tag{2}$$

until convergence. In `PowerMethod.java`, the method `initVector()` initializes all elements of vector $\pi^{(0)}$ to $1/N$. The method `multiply` computes the product of vector and a matrix.

- In `PowerMethod.java`, start by completing the method `googleMatrix` using Equation (1).
- Complete the method `compute` to implement the power iteration method, as per Equation (2). Your algorithm should stop when the mean square error (MSE) between two successive vectors $\pi^{(k)}$ and π^{k-1} is less than `PowerMethod.TOLERANCE`. You can use the method `mse` for this. As usual, take advantage of the unit tests.
- Notice that $\pi^{(k)} = \pi^{(k-1)} G$ can also be stated as $\pi^{(k)} = \pi^{(0)} G^k$. Why do we iteratively multiply the matrix with the vector instead of raising the matrix to some power? *Hint:* compare the running times.
- Now that we have a fairly efficient algorithm, we can apply it to the *Wikipedia for Schools* dataset (`wikipedia.graph`.) The articles can be viewed online[4]. How do your interpret the resulting PageRank scores?

4 Gaming the System

In this final section of the lab, we are interested in coming up with strategies to increase the PageRank of a certain page by adding edges to the graph. We consider two different scenarios:

1. We are just allowed to add incoming edges to the page which we want to increase the PageRank score.
2. We are allowed to add edges in any direction, anywhere in the graph.

Looking at Figure 1, how would you add one edge towards node 5 so as to increase its PageRank score as much as possible? Can you come up with a metric that indicates the value of a potential new edge?

- In the class `Manipulation.java`, complete the method `addIncomingEdges` and try to maximize the page rank of the *History of mathematics* page (ID: 2463) by adding at most 300 incoming edges. How do you compare to our reference score?
- Now, you are allowed to add edges *anywhere* in the graph. Does this improve the situation? Can you beat your previous, more constrained strategy? Based on these new relaxed constraints complete the method `addEdges`. We reserve the right to give a bonus to teams who demonstrate a particularly effective approach.

[3] Can you think of other ways to solve this? What tools from linear algebra would you use?

[4] Example: `http://schools-wikipedia.org/wp/u/United_States.htm`, the URL is easy to guess from the article name.