CIS 833 – Information Retrieval and Text Mining

Lecture 17

Link Analysis

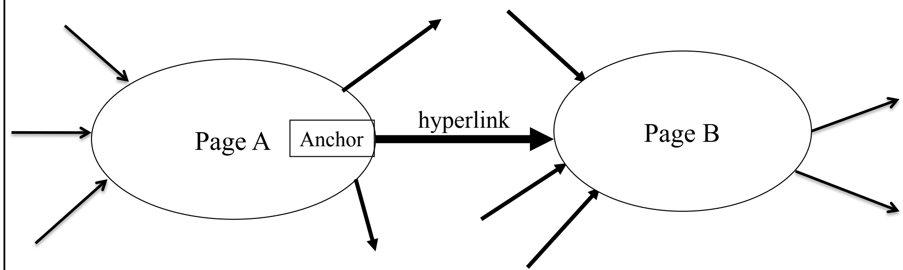
October 29, 2015

Credits for slides: Allan, Arms, Manning, Lund, Noble, Page.

Next

- Web Search
  - Textbook Chapter 21 – Web analysis
  - Monika R. Henzinger, Hyperlink Analysis for the Web. IEEE Internet Computing, vol. 5, no. 1, pp. 45-50, Jan/Feb., 2001.

The Web as a Directed Graph



Assumptions:

A hyperlink between pages denotes
- author perceived relevance (quality signal) and/or
- similar topic

The anchor of the hyperlink
- describes the target page (textual context)

Connectivity-Based Ranking

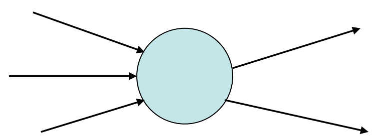
Ranking based on hyperlink analysis

- Query-independent ranking
  - PageRank: authorities
- Query-dependent ranking
  - HITS: authorities and hubs

- *Authorities* are pages that are recognized as providing significant, trustworthy, and useful information on a topic.
- *Hubs* are index pages that provide lots of useful links to relevant content pages (topic authorities).

Query-Independent Ranking

- First generation: using link counts as simple measures of popularity.
- Two basic suggestions:
  - <u>Undirected popularity:</u>
    - Each page gets a score = the number of in-links plus the number of out-links (3+2=5).
  - <u>Directed popularity:</u>
    - Score of a page = number of its in-links (3).



Query Processing

- First retrieve all pages meeting the text query (say ***gardening tools***).
- Order these pages by their link popularity (either variant on the previous page).

Spamming Simple Popularity

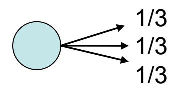
- How do you spam each of the following heuristics so your page gets a high score?
  - Each page gets a score = the number of in-links plus the number of out-links.
  - Score of a page = number of its in-links.
- How can an engine avoid such spamming?

PageRank

- Link-analysis method used by Google (Brin & Page, 1998).
- Ranks pages based on authority.
- Combating Web Spam with TrustRank
  - http://www.cs.toronto.edu/vldb04/protected/eProceedings/contents/pdf/RS15P3.PDF

PageRank Scoring – Initial Idea

- Imagine a browser doing a random walk on web pages:
  - Start at a random page
  - At each step, go out of the current page along one of the links on that page, equiprobably



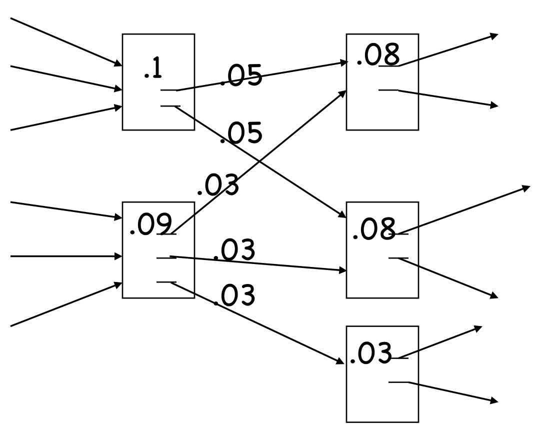
- "In the steady state" each page has a long-term visit rate - use this as the page's score.

$$R(A) = c \sum_{(B,A)\in G} R(B)/\mathrm{outdegree}(B)$$

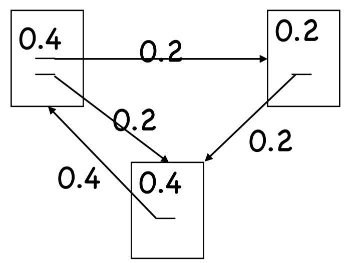
  - *c* is a normalizing constant set so that the rank of all pages always sums to 1.
- outdegree(*B*) is the number of edges leaving page *B*, that is, the number of hyperlinks on page *B*.
- A page *B* "gives" an equal fraction of its popularity to all the pages it points to (e.g., *A*).

Initial PageRank Idea

- Can view it as a process of PageRank "flowing" from pages to the pages they "cite".



Stedy-State: Sample Stable Fixpoint



Initial Algorithm

- Iterate rank-flowing process until convergence:
  Let *S* be the total set of pages.
  Initialize $\forall A \in S$: $R(A) = 1/|S| = 1/n$
  Until ranks do not change (much) (*convergence*)
  For each $A \in S$:

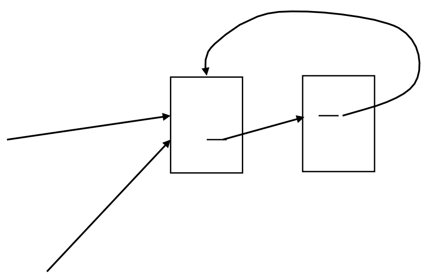
$$R'(A) = \sum_{B->A} \frac{R(B)}{out(B)}$$

$$c = 1/\sum_{A \in S} R'(A)$$

For each $A \in S$: $R(A) = cR'(A)$ (*normalize*)

Problem with Initial Idea

- The web is full of dead-ends.
  - Random walk can get stuck in dead-ends.
  - A group of pages that only point to themselves but are pointed to by other pages act as a "rank sink" and absorb all the rank in the system.
  - Makes no sense to talk about long-term visit rates.



Rank flows into
cycle and can't get out

Teleporting

- At a dead end, jump to a random web page.
- At any non-dead end, with probability 10%, jump to a random web page.
- With remaining probability (90%), go out on a random link.
  - 10% - the ε parameter

$$R(A) = c\left(\varepsilon / n + (1-\varepsilon) \sum_{(B,A)\in G} R(B)/\text{outdegree}(B)\right)$$

  - *c* is a normalizing constant set so that the rank of all pages always sums to 1.

Result of Teleporting

- Now cannot get stuck locally.
- There is a long-term rate at which any page is visited (not obvious, can be proved using Markov chains - textbook).

PageRank Algorithm

Let S be the total set of pages and n=|S|, i.e. n is the number of Web pages in the collection

Choose ε s.t. $0<\varepsilon<1$, e.g. 0.15

Initialize $\forall A \in S$: $R(A) = 1/n$

Until ranks do not change (much) (*convergence*)

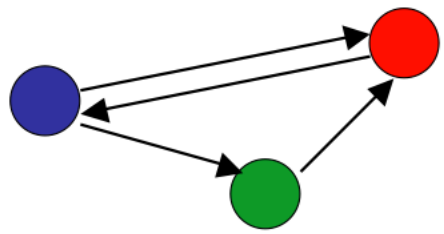
For each $A \in S$:

$$R'(A) = \left[(1-\varepsilon)\sum_{B \to A} \frac{R(B)}{out(B)}\right] + \frac{\varepsilon}{n}$$

$$c = 1/\sum_{A \in S} R'(A)$$

For each $A \in S$: $R(A) = cR'(A)$ (*normalize*)

PageRank Exercise



Random Surfer Model

- $R(A)$ models the probability that the random surfer will be on page A at any given time.
- "Jumps" are needed to prevent the random surfer from getting "trapped" in web sinks with no outgoing links.
- Markov chains are abstractions of random walks.
- **The PageRank of a page p is *the fraction of time the surfer spends at p.***

Speed of Convergence

- Early experiments on Google used 322 million links.
- PageRank algorithm converged (within small tolerance) in about 50 iterations.
- Number of iterations required for convergence is empirically O(log *n*) (where *n* is the number of links).
- Therefore calculation is quite efficient.

MapReduce Implementation

//Y is a page, PR(Y) is current PageRank of Y, and $Z_1,\ldots,Z_n$ are outgoing links from Y

Map input: $\left(Y,[PR(Y),\{Z_1,\ldots,Z_n\}]\right)$

Map output: $\left(Z_i,\dfrac{PR(Y)}{n}\right),\left(Y,\{Z_1,\ldots,Z_n\}\right)$

Reduce input: $\left(Y,[S_1,\ldots,S_m,\{Z_1,\ldots,Z_n\}]\right)$

Reduce output: $\left(Y,\left[\dfrac{\varepsilon}{N}+(1-\varepsilon)\sum_{i=1}^{m}S_m,\{Z_1,\ldots,Z_n\}\right]\right)$

MapReduce Implementation

```
public void map(LongWritable key, Text value,Context context)
{
  …
    extract page from value
    extract links from value
  …

   context.write(new Text(page), new Text(links));

   tokenize links

   while(tokenizer.hasMoreElements())
   {
       String outLink = tokenizer.nextToken().toString().trim();
       context.write(new Text(outLink), new Text(Double.toString((double)rank/(double)(totalOutLinks))));
    }
}
```