More About PageRank

Hubs and Authorities (HITS)
Combatting Web Spam
Dealing with Non-Main-Memory Web Graphs

Jeffrey D. Ullman
Stanford University
HITS

Hubs
Authorities
Solving the Implied Recursion
Mutually recursive definition:

- A *hub* links to many authorities;
- An *authority* is linked to by many hubs.

Authorities turn out to be places where information can be found.

- Example: course home pages.

Hubs tell where the authorities are.

- Example: departmental course-listing page.
Transition Matrix $A$

- HITS uses a matrix $A[i, j] = 1$ if page $i$ links to page $j$, 0 if not.
- $A^T$, the transpose of $A$, is similar to the PageRank matrix $M$, but $A^T$ has 1’s where $M$ has fractions.
- Also, HITS uses column vectors $h$ and $a$ representing the degrees to which each page is a hub or authority, respectively.
- Computation of $h$ and $a$ is similar to the iterative way we compute PageRank.
Example: H&A Transition Matrix

Yahoo

Amazon

M’soft

\[
A = \begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & 0 & 1 & \\
0 & 1 & 0 & \\
\end{pmatrix}
\]
Powers of $A$ and $A^T$ have elements whose values grow exponentially with the exponent, so we need scale factors $\lambda$ and $\mu$.

Let $h$ and $a$ be column vectors measuring the “hubbiness” and authority of each page.

Equations: $h = \lambda A a; \quad a = \mu A^T h$.

- **Hubbiness** = scaled sum of authorities of successor pages (out-links).
- **Authority** = scaled sum of hubbiness of predecessor pages (in-links).
Consequences of Basic Equations

- From $h = \lambda A a$; $a = \mu A^T h$ we can derive:
  - $h = \lambda \mu A^T A h$
  - $a = \lambda \mu A^T a$

- Compute $h$ and $a$ by iteration, assuming initially each page has one unit of hubbiness and one unit of authority.
  - Pick an appropriate value of $\lambda \mu$. 
Remember: it is only the direction of the vectors, or the relative hubbiness and authority of Web pages that matters.

As for PageRank, the only reason to worry about scale is so you don’t get overflows or underflows in the values as you iterate.
Example: Iterating H&A

\[ a = \lambda \mu A^T A \ a; \ h = \lambda \mu A A^T h \]

\[
A = \begin{bmatrix}
  1 & 1 & 1 \\
  1 & 0 & 1 \\
  0 & 1 & 0
\end{bmatrix}
\quad A^T = \begin{bmatrix}
  1 & 1 & 0 \\
  1 & 0 & 1 \\
  1 & 1 & 0
\end{bmatrix}
\quad AA^T = \begin{bmatrix}
  3 & 2 & 1 \\
  2 & 2 & 0 \\
  1 & 0 & 1
\end{bmatrix}
\quad A^T A = \begin{bmatrix}
  2 & 1 & 2 \\
  1 & 2 & 1 \\
  2 & 1 & 2
\end{bmatrix}
\]

\[
a(\text{yahoo}) = 1 \quad 5 \quad 24 \quad 114 \quad \ldots \quad 1 + \sqrt{3}
\]
\[
a(\text{amazon}) = 1 \quad 4 \quad 18 \quad 84 \quad \ldots \quad 2
\]
\[
a(\text{m’soft}) = 1 \quad 5 \quad 24 \quad 114 \quad \ldots \quad 1 + \sqrt{3}
\]

\[
h(\text{yahoo}) = 1 \quad 6 \quad 28 \quad 132 \quad \ldots \quad 1.000
\]
\[
h(\text{amazon}) = 1 \quad 4 \quad 20 \quad 96 \quad \ldots \quad 0.735
\]
\[
h(\text{microsoft}) = 1 \quad 2 \quad 8 \quad 36 \quad \ldots \quad 0.268
\]
Solving HITS in Practice

- Iterate as for PageRank; don’t try to solve equations.
- But keep components within bounds.
  - **Example**: scale to keep the largest component of the vector at 1.
  - Consequence is that $\lambda$ and $\mu$ actually vary as time goes on.
Correct approach: start with $h = [1,1,...,1]$; multiply by $A^T$ to get first $a$; scale, then multiply by $A$ to get next $h$, and repeat until approximate convergence.

You may be tempted to compute $AA^T$ and $A^TA$ first, then iterate multiplication by these matrices, as for PageRank.

Bad, because these matrices are not nearly as sparse as $A$ and $A^T$. 
Web Spam

Term Spamming
Link Spamming
What Is Web Spam?

- **Spamming** = any deliberate action solely in order to boost a Web page’s position in search-engine results.
- **Spam** = Web pages that are the result of spamming.
- SEO industry might disagree!
  - **SEO** = search engine optimization
Web Spam Taxonomy

- **Boosting** techniques.
  - Techniques for achieving high relevance/importance for a Web page.
- **Hiding** techniques.
  - Techniques to hide the use of boosting from humans and Web crawlers.
- **Term spamming.**
  - Manipulating the text of web pages in order to appear relevant to queries.

- **Link spamming.**
  - Creating link structures that boost PageRank.
Term-Spamming Techniques

- **Repetition** of terms, e.g., “Viagra,” in order to subvert TF.IDF-based rankings.
- **Dumping** = adding large numbers of words to your page.
  - **Example:** run the search query you would like your page to match, and add copies of the top 10 pages.
  - **Example:** add a dictionary, so you match every search query.
- **Key hiding technique:** words are hidden by giving them the same color as the background.
Link Spam

Design of a Spam Farm
TrustRank
Spam Mass
PageRank prevents spammers from using term spam to fool a search engine.

- While spammers can still use the techniques, they cannot get a high-enough PageRank to be in the top 10.

- Spammers now attempt to fool PageRank by *link spam* by creating structures on the Web, called *spam farms*, that increase the PageRank of undeserving pages.
Three kinds of Web pages from a spammer’s point of view:

1. *Own pages.*
   - Completely controlled by spammer.

2. *Accessible pages.*
   - E.g., Web-log comment pages: spammer can post links to his pages.

3. *Inaccessible pages.*
   - Everything else.
Spammer’s goal:
- Maximize the PageRank of target page $t$.

Technique:
1. Get as many links as possible from accessible pages to target page $t$.
2. Construct a spam farm to get a PageRank-multiplier effect.
**Goal**: boost PageRank of page $t$. One of the most common and effective organizations for a spam farm.

Note links are 2-way. Page $t$ links to all $M$ pages and they link back.
Suppose rank from accessible pages = $x$.
PageRank of target page = $y$.
Taxation rate = $1 - \beta$.
Rank of each “farm” page = $\frac{\beta y}{M} + \frac{(1 - \beta)}{N}$.

Share of “tax”;
N = size of the Web.
Total PageRank = 1.

From $t$; $M$ = number of farm pages
Analysis – (2)

\[
y = x + \beta M \left[ \frac{\beta y}{M} + \frac{(1-\beta)}{N} \right] + \frac{(1-\beta)}{N}
\]

\[
y = x + \beta^2 y + \beta \frac{(1-\beta)M}{N}
\]

\[
y = x/(1-\beta^2) + cM/N \quad \text{where} \quad c = \frac{\beta}{(1+\beta)}
\]

PageRank of each “farm” page

Inaccessible

Accessible

Own

Tax share for \( t \). Very small; ignore.
Analysis – (3)

- \( y = \frac{x}{1-\beta^2} + \frac{cM}{N} \) where \( c = \frac{\beta}{1+\beta} \).
- For \( \beta = 0.85 \), \( \frac{1}{1-\beta^2} = 3.6 \).
  - Multiplier effect for “acquired” page rank.
- By making \( M \) large, we can make \( y \) almost as large as we want.
If you design your spam farm just as was described, Google will notice it and drop it from the Web.

More complex designs might be undetected, but SEO innovations can be tracked by Google et al.

Fortunately, there are other techniques that do not rely on direct detection of spam farms.
Detecting Link Spam

- Topic-specific PageRank, with a set of “trusted” pages as the teleport set is called TrustRank.
- \( \text{Spam Mass} = \frac{\text{PageRank} - \text{TrustRank}}{\text{PageRank}}. \)
  - High spam mass means most of your PageRank comes from untrusted sources – you may be link-spam.
Two conflicting considerations:

- Human may have to inspect each trusted page, so this set should be as small as possible.
- Must ensure every “good page” gets adequate TrustRank, so all good pages should be reachable from the trusted set by short paths.
  - Implies that the trusted set must be geographically diverse, hence large.
1. Pick the top $k$ pages by PageRank.
   - It is almost impossible to get a spam page to the very top of the PageRank order.
2. Pick the home pages of universities.
   - Domains like .edu are controlled.
   - Notice that both these approaches avoid the requirement for human intervention.
Efficiency Considerations for PageRank

Multiplication of Huge Vector and Matrix

Representing Blocks of a Stochastic Matrix
Google computes the PageRank of a trillion pages (at least!).

The PageRank vector of double-precision reals requires 8 terabytes.
- And another 8 terabytes for the next estimate of PageRank.
The matrix of the Web has two special properties:

1. It is very sparse: the average Web page has about 10 out-links.

2. Each column has a single value – 1 divided by the number of out-links – that appears wherever that column is not 0.
Trick: for each column, store \( n = \) the number of out-links and a list of the rows with nonzero values \((1/n)\).

Thus, the matrix of the Web requires at least

\[
(4\times1 + 8\times10)\times10^{12} = 84 \text{ terabytes.}
\]

Integer \( n \)  
Average 10 links/column, 8 bytes per row number.
The Solution: Striping

- Divide the current and next PageRank vectors into $k$ *stripes* of equal size.
  - Each stripe is the components in some consecutive rows.
- Divide the matrix into squares whose sides are the same length as one of the stripes.
- Pick $k$ large enough that we can fit a stripe of each vector and a block of the matrix in main memory at the same time.
  - Note: the multiplication may actually be done at many machines in parallel.
At one time, we need $w_i$, $v_j$, and $M_{ij}$ in memory.

Vary $v$ slowest: $w_1 = M_{11} v_1$; $w_2 = M_{21} v_1$; $w_3 = M_{31} v_1$; $w_1 += M_{12} v_2$; $w_2 += M_{22} v_2$; $w_3 += M_{32} v_2$; $w_1 += M_{13} v_3$; $w_2 += M_{23} v_3$; $w_3 += M_{33} v_3$
Each column of a block is represented by:

1. The number $n$ of nonzero elements in the entire column of the matrix (i.e., the total number of out-links for the corresponding Web page).

2. The list of rows of that block only that have nonzero values (which must be $1/n$).

I.e., for each column, we store $n$ with each of the $k$ blocks and the out-link with whatever block has the row to which the link goes.
Representing a Block – (2)

- Total space to represent the matrix = 
  \[(4\times k + 8 \times 10) \times 10^{12} = 4k + 80\text{ terabytes.}\]

Integer n for a column is represented in each of k blocks.

Average 10 links/column, 8 bytes per row number, spread over k blocks.
We are not just multiplying a matrix and a vector.

We need to multiply the result by a constant to reflect the “taxation.”

We need to add a constant to each component of the result \( w \).

Neither of these changes are hard to do.

- After computing each component \( w_i \) of \( w \), multiply by \( \beta \) and then add \((1-\beta)/N\).
The strategy described can be executed on a single machine.

But who would want to?

There is a simple MapReduce algorithm to perform matrix-vector multiplication.

- But since the matrix is sparse, better to treat it as a relational join.
Another approach is to use many jobs, each to multiply a row of matrix blocks by the entire \( \mathbf{v} \).

Use main memory to hold the one stripe of \( \mathbf{w} \) that will be produced.

Read one stripe of \( \mathbf{v} \) into main memory at a time.

Read the block of \( \mathbf{M} \) that needs to multiply the current stripe of \( \mathbf{v} \), a tiny bit at a time.

Works as long as \( k \) is large enough that stripes fit in memory.

\( \mathbf{M} \) read once; \( \mathbf{v} \) read \( k \) times, among all the jobs.

- OK, because \( \mathbf{M} \) is much larger than \( \mathbf{v} \).
Main Memory for job $i$
Main Memory for job $i$
Main Memory for job $i$