Link Analysis

CSE 454 Advanced Internet Systems
University of Washington

Ranking Search Results

• TF / IDF Calculation
• Tag Information
  – Title, headers
• Font Size / Capitalization
• Anchor Text on Other Pages
• Link Analysis
  – HITS – (Hubs and Authorities)
  – PageRank

Authority and Hub Pages (1)

• A page is a good authority
  (with respect to a given query)
  if it is pointed to by many good hubs
  (with respect to the query).

• A page is a good hub page
  (with respect to a given query)
  if it points to many good authorities
  (for the query).

• Good authorities & hubs reinforce

Authority and Hub Pages (2)

• Authorities and hubs for a query tend to form a
  bipartite subgraph of the web graph.

Stability

• Stability
  small changes to graph → small changes to weights.

• Conclusion
  HITS is not stable.
  But PageRank is quite stable!

Pagerank Intuition

Think of Web as a big graph.

Suppose surfer keeps randomly clicking on the links.

Importance of a page = probability of being on the page

Derive transition matrix from adjacency matrix

Suppose ∃ N forward links from page P
Then the probability that surfer clicks on any one is 1/N
Matrix Representation

Let \( M \) be an \( N \times N \) matrix

\[
m_{uv} = \frac{1}{N_v} \text{ if page } v \text{ has a link to page } u \\
m_{uv} = 0 \text{ if there is no link from } v \text{ to } u
\]

Let \( R_0 \) be the initial rank vector

Let \( R_i \) be the \( N \times 1 \) rank vector for \( i \)th iteration

Then \( R_i = M \times R_{i-1} \)

### Problem: Page Sinks.

- Sink = node (or set of nodes) with no out-edges.
- Why is this a problem?

### Solution to Sink Nodes

Let:

\[ (1-c) = \text{chance of random transition from a sink.} \]

\[ N = \text{the number of pages} \]

\[
K = \begin{bmatrix}
\ldots & \frac{1}{N} & \ldots \\
\end{bmatrix}
\]

\[ M^* = cM + (1-c)K \]

\[ R_i = M^* \times R_{i-1} \]

### Computing PageRank - Example

\[
M = \begin{bmatrix}
A & B & C & D \\
0 & 0 & 0 & \frac{1}{2} \\
0 & 0 & 0 & \frac{1}{2} \\
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

\[
R_0 = \begin{bmatrix}
0.05 & 0.05 & 0.05 & 0.45 \\
0.05 & 0.05 & 0.05 & 0.45 \\
0.85 & 0.85 & 0.05 & 0.05 \\
0.05 & 0.05 & 0.85 & 0.05 \\
\end{bmatrix}
\]

\[
R_{30} = \begin{bmatrix}
0.176 & 0.176 & 0.332 & 0.316 \\
\end{bmatrix}
\]

### Another Example (Details)

Note: assume \( 1/N = 0.15 \)

Instead on \( R_0 = \) uniform \( 1/N \)

let \( R_0 = 1 \) everywhere

### Navigational Effects (matrix M)
Random Jumps Give Extra 0.15
Page A: 0.85 (from Page C) + 0.15 (random) = 1
Page B: 0.425 (from Page A) + 0.15 (random) = 0.575
Page C: 0.85 (from Page D) + 0.85 (from Page B) + 0.425 (from Page A) + 0.15 (random) = 2.275
Page D: receives nothing but 0.15 (random) = 0.15

Page A
Page B
Page C
Page D

Round 2
Page A: 2.275*0.85 (from Page C) + 0.15 (random) = 2.03875
Page B: 1*0.85/2 (from Page A) + 0.15 (random) = 0.575
Page C: 0.15*0.85 (from D) + 0.575*0.85 (from B) + 1*0.85/2 (from Page A) +0.15 (random) = 1.1925
Page D: receives nothing but random 0.15 = 0.15

Page A
Page B
Page C
Page D

Example of calculation (4)
After 20 iterations, we get

Page A
Page B
Page C
Page D

Example - Conclusions
• Page C has highest importance in page graph!
  – Page A has the next highest:
• Convergence requires
  – Many iterations
  – Is it guaranteed??

Linear Algebraic Interpretation
• PageRank = principle eigenvector of $M^*$
  – in limit
• HITS = principle eigenvector of $M^* \times (M^*)^T$
  – Where $[ \cdot]^T$ denotes transpose $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$
• Can prove PageRank is stable
• And HITS isn’t

Stability Analysis (Empirical)
• Make 5 subsets by deleting 30% randomly

PageRank much more stable
Practicality

- **Challenges**
  - M no longer sparse (don’t represent explicitly!)
  - Data too big for memory (be sneaky about disk usage)
- **Stanford Version of Google**:
  - 24 million documents in crawl
  - 147GB documents
  - 259 million links
  - Computing pagerank “few hours” on single 1997 workstation
- **But How?**
  - Next discussion from Haveliwala paper…

Efficient Computation: Preprocess

- **Remove ‘dangling’ nodes**
  - Pages w/ no children
- **Then repeat process**
  - Since now more danglers
- **Stanford WebBase**
  - 25 M pages
  - 81 M URLs in the link graph
  - After two prune iterations: 19 M nodes

Representing ‘Links’ Table

- **Stored on disk in binary format**

<table>
<thead>
<tr>
<th>Source node</th>
<th>Outdegree</th>
<th>Destination nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12, 26, 58, 94</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5, 56, 69</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1, 9, 10, 36, 78</td>
</tr>
</tbody>
</table>

- **Size for Stanford WebBase: 1.01 GB**
  - Assumed to exceed main memory

Defining PageRank

Let u be a web page,
- \( F_u \) = set of pages u points (forward) to,
- \( B_u \) = set of pages that point to u (i.e. from behind),
- \( N_u = |F_u| \) = the out-degree of u.

The rank (importance) of page u … (first cut):

\[
R(u) = \sum_{v \in B_u} \frac{R(v)}{N_v}
\]

Compute Iteratively:

\[
R_i(u) = \sum_{v \in B_u} \frac{R_{i-1}(v)}{N_v}
\]

Algorithm 1

\[
\forall s \text{ Source}[s] = 1/N
\]

\[
\text{while residual} > \tau \{
\forall \text{ Dest}[d] = 0
\text{while not Links.eof()} \{
\text{Links.read(source, n, dest1, … destn)}
\text{for } j = 1, \ldots, n
\text{Dest[destj]} = \text{Dest[destj]} + \text{Source[source]/n}
\}
\forall \text{ Dest}[d] = (1-c) \times \text{Dest}[d] + c/N \quad \text{/* damping c= 1/N */}
\text{residual} = |\text{Source} – \text{Dest}| \quad \text{/* recompute every few iterations */}
\text{Source = Dest}
\}

Analysis

- **If memory can hold both source & dest**
  - IO cost per iteration is |Links|
  - Fine for a crawl of 24 M pages
  - But web > 8 B pages in 2005 [Google]
  - Increase from 320 M pages in 1997 [NEC study]
- **If memory only big enough to hold just dest...?**
  - Sort Links on source field
  - Read Source sequentially during rank propagation step
  - Write Dest to disk to serve as Source for next iteration
  - IO cost per iteration is |Source| + |Dest| + |Links|
- **If memory can’t even hold dest**
  - Random access pattern will make working set = |Dest|
  - Thrash!!!
Block-Based Algorithm

- Partition Dest into B blocks of D pages each
  - If memory = P physical pages
  - D < P-2 since need input buffers for Source & Links

- Partition (sorted) Links into B files
  - Links only has some of the dest nodes for each source
  - Links only has dest nodes such that
    - \( DD^i \leq \text{dest} < DD^{(i+1)} \)
    - Where \( DD = \) number of 32 bit integers that fit in D pages

Analysis of Block Algorithm

- IO Cost per iteration =
  - \( B \times |\text{Source}| + |\text{Dest}| + |\text{Links}|(1+e) \)
  - e is factor by which Links increased in size
    - Typically 0.1-0.3
    - Depends on number of blocks
  - Algorithm ~ nested-loops join

Partitioned Link File

Comparing the Algorithms

Adding PageRank to a SearchEngine

- Weighted sum of importance+similarity with query
- \( \text{Score}(q, d) = w \times \text{sim}(q, p) + (1-w) \times R(p), \) if \( \text{sim}(q, p) > 0 \)
  = 0, otherwise
- Where
  - \( 0 < w < 1 \)
  - \( \text{sim}(q, p), R(p) \) must be normalized to \([0, 1]\).
Summary of Key Points

• PageRank Iterative Algorithm
• Sink Pages
• Efficiency of computation – Memory!
  – Don’t represent M* explicitly.
  – Minimize IO Cost.
  – Break arrays into Blocks.
  – Single precision numbers ok.
• Number of iterations of PageRank.
• Weighting of PageRank vs. doc similarity.