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

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 2 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×N matrix
m_u,v = 1/N_v if page v has a link to page u
m_u,v = 0 if there is no link from v to u
Let R_0 be the initial rank vector
Let R_i be the N×1 rank vector for i^th iteration
Then R_i = M × 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) = chance of random transition from a sink.
N = the number of pages
K = 

M' = cM + (1-c)K
R_i = M' × R_{i-1}
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.

(A page can be a good authority and a good hub)

Linear Algebraic Interpretation

- PageRank = principle eigenvector of $M^*$ (in limit)
- HITS = principle eigenvector of $M^* \times (M^*)^T$
  - Where $[ ]^T$ denotes transpose
  
  \[
  \begin{bmatrix}
  1 \\
  3 \\
  2 \\
  4
  \end{bmatrix}
  \]

- Stability
  Small changes to graph $\Rightarrow$ small changes to weights.
  - Can prove PageRank is stable
  - And HITS isn’t

Stability Analysis (Empirical)

- Make 5 subsets by deleting 30% randomly

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>12</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>52</td>
<td>20</td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>171</td>
<td>119</td>
<td>99</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>135</td>
<td>56</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>179</td>
<td>159</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>316</td>
<td>141</td>
<td>170</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>257</td>
<td>107</td>
<td>72</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>170</td>
<td>80</td>
<td>69</td>
<td>18</td>
</tr>
</tbody>
</table>

- 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 (32 bit int)</th>
<th>Outdegree (16 bit int)</th>
<th>Destination nodes (32 bit integers)</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>

- **Stanford WebBase: 1.01 GB**
  - Assumed to exceed main memory
  - (But source & dest assumed to fit)

Algorithm 1

```
∀ Source[s] = 1/N
while residual > τ {
    ∀ Dest[d] = 0
    while not Links.eof() {
        Links.read(source, n, dest1, … dest n)
        for j = 1… n
    }
    ∀ Dest[d] = (1-c) * Dest[d] + c/N       /* dampening  c= 1/N */
    residual = ||Source – Dest|| /* recompute every few iterations */
    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|
- **But What 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
  - Specifically, Links only has dest nodes such that
    - DD*i <= dest < DD*(i+1)
  - Where DD = number of 32 bit integers that fit in D pages

Partitioned Link File

<table>
<thead>
<tr>
<th>Source node (32 bit int)</th>
<th>Outdegr (16 bit int)</th>
<th>Num out (16 bit)</th>
<th>Destination nodes (32 bit int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>2</td>
<td>12, 26</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1, 9, 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source node (32 bit int)</th>
<th>Outdegr (16 bit int)</th>
<th>Num out (16 bit)</th>
<th>Destination nodes (32 bit int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>1</td>
<td>94</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>78</td>
</tr>
</tbody>
</table>
Analysis of Block Algorithm

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

Comparing the Algorithms

Adding PageRank to a SearchEngine

- Weighted sum of importance+similarity with query
- Score(q, d) = \( w \cdot \text{sim}(q, p) + (1-w) \cdot R(p) \), 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.