Announcements (December 3)

- Homework #4 sample solution available
- Course project demo period continues!
- Final exam next Tuesday, Dec. 8, 9am-12pm
  - Again, open book, open notes
  - Focus on the second half of the course
  - Sample final solution available
- I will be running a final review session next Monday, Dec. 7, 4-5pm

Keyword search

What are the documents containing both "database" and "search"?
Keywords × documents

<table>
<thead>
<tr>
<th>Keywords</th>
<th>All documents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Document 1</td>
</tr>
<tr>
<td>&quot;cat&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;database&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;dog&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;search&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

1 means keyword appears in the document; 0 means otherwise

- Inverted lists: store the matrix by rows
- Signature files: store the matrix by columns

Inverted lists

- Store the matrix by rows
- For each keyword, store an inverted list
  - *(keyword, doc-id-list)*
  - *("database", {3, 7, 142, 857, …})*
  - *("search", {3, 9, 192, 512, …})*
  - It helps to sort doc-id-list (why?)
- Vocabulary index on keywords
  - B*-tree, hash-based, or trie (later)
- How large is an inverted list index?

Using inverted lists

- Documents containing "database"
  - Use the vocabulary index to find the inverted list for "database"
  - Return documents in the inverted list
- Documents containing "database" AND "search"
  - Return documents in the intersection of the two inverted lists
- OR? NOT?
What are “all” the keywords?

- All sequences of letters (up to a given length)?
  - … that actually appear in documents!
- All words in English?
- Plus all phrases?
  - Alternative: approximate phrase search by proximity
- Minus all stop words
  - They appear in nearly every document, e.g., a, of, the, it
  - Not useful in search
- Combine words with common stems
  - Example: database, databases
  - They can be treated as the same for the purpose of search

Frequency and proximity

- Frequency
  - \(\text{keyword}, \{\langle \text{doc-id}, \text{number-of-occurrences}\rangle, \langle \text{doc-id}, \text{number-of-occurrences}\rangle, \ldots\}\)\)
- Proximity (and frequency)
  - \(\text{keyword}, \{\langle \text{doc-id}, \{\langle \text{position-of-occurrence}\rangle, \langle \text{position-of-occurrence}\rangle, \ldots\\}\rangle, \langle \text{doc-id}, \{\langle \text{position-of-occurrence}\rangle, \ldots\\}\rangle, \ldots\}\)\)
  - When doing AND, check for positions that are near

Signature files

- Store the matrix by columns and compress them
- For each document, store a \(w\)-bit signature
- Each word is hashed into a \(w\)-bit value, with only \(s < w\) bits turned on
- Signature is computed by taking the bit-wise OR of the hash values of all words on the document

\[
\text{hash}(\text{"database"}) = 0110 \\
\text{hash}(\text{"dog"}) = 1100 \\
\text{hash}(\text{"cat"}) = 0010 \\
\]

- \(\text{doc}_1\) contains "database": 0110
- \(\text{doc}_2\) contains "dog": 1100
- \(\text{doc}_3\) contains "cat" and "dog": 1110

- Some false positives; no false negatives
Bit-sliced signature files

- Motivation
  - To check if a document contains a word, we only need to check the bits that are set in the word's hash value.
  - So why bother retrieving all w bits of the signature?
- Instead of storing n signature files, store w bit slices.
- Only check the slices that correspond to the set bits in the word's hash value.
- Start from the sparse slices.

Inverted lists versus signatures

- Inverted lists better for most purposes (TODS, 1998)
- Problems of signature files
  - False positives
  - Hard to use because r, w, and the hash function need tuning to work well
  - Long documents will likely have mostly 1's in signatures
  - Common words will create mostly 1's for their slices
  - Difficult to extend with features such as frequency, proximity
- Saving grace of signature files

Ranking result pages

- A single search may return many pages
  - A user will not look at all result pages
  - Complete result may be unnecessary
- Possible ranking criteria
  - Based on content
    - Number of occurrences of the search terms
    - Similarity to the query text
  - Based on link structure
    - Backlink count
    - PageRank
  - And more…
Textual similarity

- Terms \( \{t_1, \ldots, t_n\} \) and documents \( D = \{d_1, d_2, \ldots\} \)
- IDF (Inverse Document Frequency) of \( t_i \):
  \[
  idf_i = \log \left( \frac{\text{# of docs in } D \text{ containing } t_i}{|D|} \right)
  \]
- TF (Term Frequency) of \( t_i \) in \( d_j \):
  \[
  tf_{i,j} = \frac{(\text{# of times } t_i \text{ appears in } d_j)}{(\text{# of term appearances in } d_j)}
  \]
- TF-IDF weight vector of \( d_j \):
  \[
  w_j = (tf_{1,j}idf_1, \ldots, tf_{n,j}idf_n)
  \]
- Textual similarity between two docs \( d_j \) and \( d_k \) can be measured by the normalized dot product of these vectors, i.e.:
  \[
  \frac{(w_j \cdot w_k)}{\|w_j\| \cdot \|w_k\|} = \frac{\sum_i tf_{i,j}idf_i \cdot tf_{i,k}idf_k}{\sqrt{\sum_i tf_{i,j}^2idf_i} \cdot \sqrt{\sum_i tf_{i,k}^2idf_k}}
  \]

Why weigh significance by IDF?

Problems with content-based ranking

- Many pages containing search terms may be of poor quality or irrelevant
  - Example: a page with just a line “search engine”
- Many high-quality or relevant pages do not even contain the search terms
  - Example: Google homepage
- Page containing more occurrences of the search terms are ranked higher; spamming is easy
  - Example: a page with line “search engine” repeated many times
Backlink

- A page with more backlinks is ranked higher
- Intuition: Each backlink is a "vote" for the page’s importance

Google’s PageRank

- Main idea: Pages pointed by high-ranking pages are ranked higher
  - Definition is recursive by design
  - Based on global link structure; hard to spam
- Naïve PageRank
  - $F(p)$: set of pages that page $p$ points to
  - $B(p)$: set of pages that point to $p$
  - $\text{PageRank}(p) = \sum_{q \in B(p)} (\text{PageRank}(q) / |F(q)|)$
    - Each page gets a boost from every page pointing to it
    - Each page distributes its importance evenly to pages it points to

Calculating naïve PageRank

- Initially, set all PageRank’s to 1; then evaluate
  $\text{PageRank}(p) \leftarrow \sum_{q \in B(p)} (\text{PageRank}(q) / N(q))$
  repeatedly until the values converge (i.e. a fixed point is reached)
Random surfer model

- A random surfer
  - Starts with a random page
  - Randomly selects a link on the page to visit next
  - Never uses the “back” button

- PageRank($p$) measures the probability that a random surfer visits page $p$

Problems with the naïve PageRank

- Dead end: a page with no outgoing links
  - A dead end causes all importance to “leak” eventually out of the Web

- Spider trap: a group of pages with no links out of the group
  - A spider trap will eventually accumulate all importance of the Web

Practical PageRank

- $d$: decay factor
- \[ \text{PageRank}(p) = d \cdot \sum_{q \in B(p)} \left( \frac{\text{PageRank}(q)}{N(q)} \right) + (1 - d) \]

- Intuition in the random surfer model
  - A surfer occasionally gets bored and jumps to a random page on the Web instead of following a random link on the current page
Google (1998)

- Inverted lists in practice contain a lot of context information

<table>
<thead>
<tr>
<th>HIT 2 bytes</th>
<th>Relative font size</th>
<th>Capitalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>plan cap</td>
<td>1</td>
<td>cap</td>
</tr>
<tr>
<td>flippy cap</td>
<td>1</td>
<td>cap</td>
</tr>
<tr>
<td>flippy cap</td>
<td>4</td>
<td>position 1x</td>
</tr>
<tr>
<td>in the page</td>
<td></td>
<td>within the page</td>
</tr>
</tbody>
</table>

- PageRank is not the final ranking
  - Type-weight: depends on the type of the occurrence
    - For example, large font weights more than small font
  - Count-weight: depends on the number of occurrences
    - Increases linearly first but then tapers off
  - For multiple search terms, nearby occurrences are matched together and a proximity measure is computed
    - Closer proximity weights more

Trie: a string index

- A tree with edges labeled by characters
- A node represents the string obtained by concatenating all characters along the path from the root

![Trie Diagram](image)

- Compact trie: replace a path without branches by a single edge labeled by a string

Suffix tree

- Index all suffixes of a large string in a compact trie
- Can support arbitrary substring matching
- Internal nodes have fan-out $\geq 2$ (except the root)
- No two edges out of the same node can share the same first character

To get linear space
- Instead of inlining the string labels, store pointers to them in the original string
- Bad for external memory
Patricia trie, Pat tree, String B-tree

A Patricia trie is just like a compact trie, but
- Instead of labeling each edge by a string, only label by the first character and the string length
- Leaves point to strings
- Faster search (especially for external memory) because of inlining of the first character
- But must validate answer at leaves for skipped characters

- A Pat tree indexes all suffixes of a string in a Patricia trie
- A String B-tree uses a Patricia trie to store and compare strings in B-tree nodes

Summary

- General tree-based string indexing tricks
  - Trie, Patricia trie, String B-tree
- Two general ways to index for substring queries
  - Index words: inverted lists, signature files
  - Index all suffixes: suffix tree, Pat tree, suffix array (not covered)
- Web search and information retrieval go beyond substring queries
  - TF-IDF, PageRank, …