15-853: Algorithms in the Real World

Indexing and Searching I
  - Introduction
  - Inverted Indices

Outline for next few classes

Inverted Indices (used by all search engines)
  - Compression
  - The lexicon
  - Merging terms (unions and intersections)

Vector Models

Latent Semantic Indexing

Link Analysis:
  - PageRank (Google)
  - HITS

Duplicate Removal

Basic Model

"Document Collection"

Index ← Query

Document List

Applications:
  - Web, mail and dictionary searches
  - Law and patent searches
  - Information filtering (e.g., NYT articles)

Goal: Speed, Space, Accuracy, Dynamic Updates

How big is an Index?

<table>
<thead>
<tr>
<th>Month</th>
<th>Millions of Web Pages Indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec 2001</td>
<td>500,000</td>
</tr>
</tbody>
</table>

Dec 2001, self proclaimed sizes (gg = google)
Source: Search Engine Watch
**Main Approaches**

- **Full text searching**
  - e.g. grep, agrep (used by many mailers)
- **Inverted Indices**
  - good for short queries
  - used by most search engines
- **Signature files**
  - good for longer queries with many terms
- **Vector Space Models**
  - good for better accuracy
  - used in clustering, SBD

**Queries**

- **Types of queries on multiple terms**
  - boolean (and, or, not, andnot)
  - proximity (adj, within nW)
  - keyword sets
  - in relation to other documents
- **And within each term**
  - prefix matches
  - wildcards
  - edit distance bounds

**Technique used Across Methods**

- **Case folding**
  - London → London
- **Stemming**
  - compress = compression = compressed
  - (several off-the-shelf English Language stemmers are freely available)
- **Stop words**
  - to, the, it, be, or, is
  - how about “to be or not to be”
- **Thesaurus**
  - fast → rapid

**Documents as Bipartite Graph**

- **Called an Inverted File Index**
  - can be stored using adjacency lists, also called
  - posting lists (or files)
  - inverted file entry
- **Example size of the Web**
  - 538 million terms
  - 200 documents
  - 333,856 edges
  - For the Web, multiply by 1 billion
Documents as Bipartite Graph

Implementation Issues:
- Space for posting lists: these take almost all the space
- Access to lexicon
  - Btrees, tries, hashing
  - Prefix and wildcard queries
- Merging posting list
  - Multiple term queries

1. Space for Posting Lists

Posting lists can be as large as the document data
- Saving space and the time to access the space is critical for performance
- We can compress the lists, but we need to uncompress on the fly

Difference encoding:

Let's say the term elephant appears in documents:
- 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0

Then the difference code is:
- 0, 1, 0, 1, 0, 0, 0, 0, 0, 0

Some Codes

Gamma code:
- If most significant bit of n is in location k, then
  \[ \gamma(n) = O^{k-1} n \] \[ 2 \log(n) - 1 \text{ bits} \]

Delta code:
- \[ \gamma(k)n \] \[ 2 \log(\log(n)) + \log(n) - 1 \text{ bits} \]

Frequency coded:
- Base on actual probabilities of each distance

Global vs. Local Probabilities

Global:
- Count \# of occurrences of each distance
- Use Huffman or arithmetic code

Local:
- Generate counts for each list
  - elephant: 3, 2, 1, 2, 53, 1
- Problem: counts take too much space
- Solution: batching
  - Group into buckets by \( \lceil \log(\text{length}) \rceil \)
Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Bits/edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>1</td>
</tr>
<tr>
<td>Binary</td>
<td>2</td>
</tr>
<tr>
<td>Gamma</td>
<td>3</td>
</tr>
<tr>
<td>Delta</td>
<td>4</td>
</tr>
<tr>
<td>Huffman</td>
<td>5</td>
</tr>
<tr>
<td>Local</td>
<td>6</td>
</tr>
<tr>
<td>Skewed Bernoulli</td>
<td>7</td>
</tr>
<tr>
<td>Batched Huffman</td>
<td>8</td>
</tr>
</tbody>
</table>

- Bits per edge based on the document collection.
- Total size: # bytes/MBytes

2. Accessing the Lexicon

- We all know how to store a dictionary, ¿U-¿
- It is best if lexicon fits in memory---can we avoid storing all characters of all words?
- What about prefix or wildcard queries?

- Some possible data structures
  - Front Coding
  - Tries
  - Perfect Hashing
  - B-trees

Prefix and Wildcard Queries

- Prefix Queries
  - Handled by all access methods except hashing
- Wildcard Queries
  - n-gram
  - Rotated lexicon
n-gram

Consider every block of n characters in a term:
* e.g. 2-gram of ðezebel "sj, je, ez, ze, eb, el, 1s"

Break wildcard query into an n-grams and search.
* e.g. j*el would
1. search for $j, el, 1s$ as if searching for documents
2. find all potential terms
3. filter matches for which the order does not match

Rotated Lexicon

Consider every rotation of a term:
* e.g. ðezebel - ðezebel, ðêzebe, el ðezeb, bel ðzez
o o store lexicon of all rotations
Given a query find longest contiguous block (i th rotation) and search for it:
* e.g. õel - õ search for el õ in lexicon
Note that each lexicon entry corresponds to a single term
* e.g. ebel õez can only mean ðezebel

3. Merging Posting Lists

Let say queries are expressions over:
- and, or, andnot
 Viel the list of documents for a term as a set:
then
- $e_1$ and $e_2 \rightarrow S_1 \cap S_2$
- $e_1$ or $e_2 \rightarrow S_1 \cup S_2$
- $e_1$ andnot $e_2 \rightarrow S_1 - S_2$

Some notes:
- the sets ordered in the "posting lists"
- $S_1$ and $S_2$ can differ in size substantially
- might be good to keep intermediate results
- persistence is important

Union, Intersection, Merging

Given two sets A and B, how long does this take?
For intersection, union and set difference:
Assume elements are taken from a total order (\$)
Very similar to merging two sets A and B, ho long does this take?

Lower bound:
- There are n elements of A and n \$ m positions in the output they could belong
- choose (n \$ m, n) possibilities
- assuming comparison based model, the decision tree has that many leaves and depth log of that
- Assuming m \$ n this give \$m log ((n \$ m)/ln))
Merging: Upper bounds

The algorithm shows $O(m \log((n/m)\ln))$ upper bounds using 2-3 trees with cross links and parent pointers. Very messy.

We will take a different approach, and base our two operations: **split** and **join**

---

**Split and Join**

**Split($S, v$)**: Split $S$ into two sets $S_1 = \{x \in S : x < v\}$ and $S_2 = \{x \in S : x \geq v\}$. Also return a flag which is true if $v \in S$.
- **Split($\emptyset$, 15, 18, 22, 18) → $\emptyset$, 15, 18, 22** True
- **Join($S_1$, $S_2$)**: Assuming $\forall k \in S_1, k_i \in S_2 : k_j \in k_i$, it returns $S_1 \cup S_2$.
- **Join($\emptyset$, 11, 10, 22, 18) → $\emptyset$, 11, 10, 22**

Time for both:
- $\Theta \log(\log(\log|S_1,| \log|S_2|))$, can be shown
- $\Theta \log|S_1|$, will suffice for us (shown later)

---

**Union with Split and Join**

**Union($S_1$, $S_2$)**
- **if empty($S_1$)** then return $S_2$.
- **else**
  - $(S_{20}, S_{20}, f) = \text{Split}(S_2, \text{first}(S_1))$
  - return $\text{Join}(S_{20}, \text{Union}(S_{20}, S_1))$

---

**Runtime of Union**

<table>
<thead>
<tr>
<th>Out</th>
<th>o1</th>
<th>o2</th>
<th>o3</th>
<th>o4</th>
<th>o5</th>
<th>o6</th>
<th>o7</th>
</tr>
</thead>
</table>

$T_{\text{union}} = O(\sum_i \log |D_i|) + \sum_i \log |D_i|)$

**Splits**

**Joins**

Since the logarithm function is concave, this is maximized when blocks are as close as possible to each size, therefore

$T_{\text{union}} = O(\sum_{i=1}^m \log [n \log (n/m) + 1])$

$= O(m \log ((n/m) \ln))$
Intersection with Split and Join

\[ \text{Intersect}(S_i, S_j) \]
\[ \begin{cases} \text{if empty}(S_i) \text{ then return } \emptyset \\ \text{else} \\ \quad (S_{22}, S_{23}, \text{flag}) = \text{Split}(S_2, \text{first}(S_1)) \\ \quad \text{if flag then} \\ \quad \quad \text{return Join}(\text{first}(S_1), \text{Intersect}(S_{22}, S_j)) \\ \quad \text{else} \\ \quad \quad \text{return Intersect}(S_{22}, S_j) \end{cases} \]

Efficient Split and Join

Recall that \( T = O(\log |S_j|) \)

How do we implement this efficiently?

Treaps

Every key is given a random priority
- keys are stored in-order
- priorities are stored in heap-order

\( (\text{key,priority}) : (1,23), (2,10), (5,11), (1,35), (12,30) \)

If the priorities are unique, the tree is unique.

Left Spinal Treap

The time to split \( L \) length from Start to split location is \( L \) in the expected case, where \( L \) is the path length between Start and the split location.

The time to Join is the same.
Analysis

\[ P_i = \text{length of path from Start to } i \]  
\[ P_j = \text{Ex}[P_j] \]  
\[ A_j = \begin{cases} 
1 & \text{if } x_i \text{ is an ancestor of } x_j \\
0 & \text{otherwise} 
\end{cases} \]  
\[ a_{ij} = \text{Ex}[A_{ij}] \]  
\[ C_{ilm} = \begin{cases} 
1 & \text{if } x_i \text{ is a common ancestor of } x_l \text{ and } x_m \\
0 & \text{otherwise} 
\end{cases} \]  
\[ c_{ilm} = \text{Ex}[C_{ilm}] \]  
\[ P_i = \sum_{j=1}^{n} A_{ij} + \sum_{j=1}^{n} (A_{ij} - C_{ilm}) \]

Analysis Continued

\[ \sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{n} \frac{1}{|i-j|+1} = \sum_{j=1}^{n} \frac{1}{i+j} < 1 + \log i \text{ (harmonic number } H_i) \]

Can similarly show that:

\[ \sum_{j=1}^{n} (a_{ij} - c_{ilm}) = O(\log i) \]

Therefore the expected path length and runtime for split and join is \( O(\log i) \).

Similar technique can be used for other properties of Treaps.

Analysis Continued

\[ \text{Ex}[P_i] = p_i = \sum_{j=1}^{n} a_{ij} + \sum_{j=1}^{n} (a_{ij} - c_{ilm}) \]

Lemma: \( a_{ij} = \frac{1}{|i-j|+1} \)

Proof:
1. \( i \) is an ancestor of \( j \) iff \( i \) has a greater priority than all elements between \( i \) and \( j \), inclusive.
2. there are \( |i-j|+1 \) such elements each with equal probability of having the highest priority.

And back to Inverted Indices

It’s shown how to take Unions and Intersections, but Treaps are not very space efficient.

Idea: if priorities are in the range \( [1, n] \) then any node with priority \( \alpha \) is stored compressed.

\( \alpha \) represents fraction of uncompressed nodes.