15-499: Algorithms and Applications

Indexing and Searching I (how google and the likes work)

Indexing and Searching Outline

- Introduction:
  - model
  - query types
  - common techniques (stop words, stemming, ...)

- Inverted Indices: Compression, Lexicon, Merging

- 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?**

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

**Precision and Recall**

Precision: \( \frac{\text{number retrieved that are relevant}}{\text{total number retrieved}} \)
Recall: \( \frac{\text{number relevant that are retrieved}}{\text{total number relevant}} \)

Typically a tradeoff between the two.

**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, SVD, ...
Queries
Types of Queries on Multiple “terms”
- boolean (and, or, not, andnot)
- proximity (adj, within <n>)
- 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, ...
how about “to be or not to be”
Thesaurus
fast -> rapid

Other Methods
Document Ranking:
Returning an ordered ranking of the results
- A priori ranking of documents (e.g. Google)
- Ranking based on “closeness” to query
- Ranking based on “relevance feedback”
Clustering and “Dimensionality Reduction”
- Return results grouped into clusters
- Return results even if query terms does not appear but are clustered with documents that do
Document Preprocessing
- Removing near duplicates
- Detecting spam

Indexing and Searching Outline
Introduction: model, query types
Inverted Indices:
- Index compression
- The lexicon
- Merging terms (unions and intersections)
Vector Models:
Latent Semantic Indexing:
Link Analysis: PageRank (Google), HITS
Duplicate Removal:
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 TREC
- 538K terms
- 742K documents
- 333,856K edges
For the web, multiply by 5-10K

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:
[3, 5, 20, 21, 23, 76, 77, 78]
then the difference code is
[3, 2, 15, 1, 2, 53, 1, 1]

Implementation Issues:
1. Space for posting lists
take almost all the space
2. Access to lexicon
- btrees, tries, hashing
- prefix and wildcard queries
3. Merging posting list
- multiple term queries

Some Codes
Gamma code:
if most significant bit of n is in location k, then
\[ \text{gamma}(n) = 0^{k-1} n[k.0] \]
2 log(n) - 1 bits
Delta code:
\[ \text{gamma}(k)n[k.0] \]
2 log(log(n)) + log(n) - 1 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, 1]
- Problem: counts take too much space
- Solution: batching
  - group into buckets by \( \lceil \log(\text{length}) \rceil \)

Performance

<table>
<thead>
<tr>
<th></th>
<th>bits/edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>20.00</td>
</tr>
<tr>
<td>Gamma</td>
<td>6.43</td>
</tr>
<tr>
<td>Delta</td>
<td>6.19</td>
</tr>
<tr>
<td>Huffman</td>
<td>5.83</td>
</tr>
<tr>
<td>Local</td>
<td></td>
</tr>
<tr>
<td>Skewed Bernoulli</td>
<td>5.28</td>
</tr>
<tr>
<td>Batched Huffman</td>
<td>5.27</td>
</tr>
</tbody>
</table>

Bits per edge based on the TREC document collection
Total size = 333M * .66 bytes = 222Mbytes

2. Accessing the Lexicon

We all know how to store a dictionary, BUT...
- 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

Front Coding

<table>
<thead>
<tr>
<th>Word</th>
<th>front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>7_jezebel</td>
<td>0,7_jezebel</td>
</tr>
<tr>
<td>5_jezer</td>
<td>4,1,r</td>
</tr>
<tr>
<td>7_jezerit</td>
<td>5,2,it</td>
</tr>
<tr>
<td>6_jeziah</td>
<td>3,3,iah</td>
</tr>
<tr>
<td>6_jeziel</td>
<td>4,2,el</td>
</tr>
<tr>
<td>7_jeziah</td>
<td>3,4,iah</td>
</tr>
</tbody>
</table>

For large lexicons can save 75% of space
But what about random access?
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 jezebel -> $j, je, ez, ze, eb, el, l$

Break wildcard query into an n-grams and search.
e.g. j*el would
1. search for $j, el, l$ as
   if searching for documents
2. find all potential terms
3. remove matches for which
   the order is incorrect

Rotated Lexicon
Consider every rotation of a term:
e.g. jezebel ->
   $jezebel, l$jezebe, el$jezeb, bel$jeze

Now store lexicon of all rotations
Given a query find longest contiguous block (with rotation)
and search for it:
e.g. j*el -> search for el$j in lexicon
Note that each lexicon entry corresponds to a single term
   e.g. ebel$jez can only mean jezebel

3. Merging Posting Lists
Lets say queries are expressions over:
- and, or, andnot
View the list of documents for a term as a set:
Then
   e₁ and e₂ -> S₁ intersect S₂
   e₁ or e₂ -> S₁ union S₂
   e₁ andnot e₂ -> S₁ diff S₂

Some notes:
- the sets ordered in the "posting lists"
- S₁ and S₂ can differ in size substantially
- might be good to keep intermediate results
- persistence is important
Union, Intersection, and Merging

Given two sets of length \( n \) and \( m \) how long does it take for intersection, union and set difference?
Assume elements are taken from a total order (\( < \)).
Very similar to merging two sets \( A \) and \( B \), how long does this take?

What is a lower bound?

Merging: Upper bounds

Brown and Tarjan show an
\( O(m \log((n+m)/n)) \) upper bound
using 2-3 trees with cross links and parent pointers. Very messy.

We will take different approach, and base on two operations: split and join

Split and Join

**Split(\( S, v \))**:
Split \( S \) into two sets
\( S_v = \{ s \in S \mid s < v \} \) and \( S_v = \{ s \in S \mid s > v \} \).
Also return a flag which is true if \( v \in S \).
- Split((7,9,15,18,22), 18) \( \rightarrow\) (7,9,15),(22),True

**Join(\( S_v, S_v \))**:
Assuming \( \forall k, e \in S_v \), \( k \) in \( S_v \), : \( k < k \),
returns \( S_v \cup S_v \).
- Join((7,9,11),(14,22)) \( \rightarrow\) (7,9,11,14,22)
Time for Split and Join

\[ \text{Split}(S, v) \rightarrow (S_L, S_R, \text{flag}) \quad \text{Join}(S_L, S_R) \rightarrow S \]

Naively:
- \( T = O(|S|) \)

Less Naively:
- \( T = O(\log |S|) \)

What we want:
- \( T = O(\log(\min(|S_L|, |S_R|))) \) -- can be shown
- \( T = O(\log |S|) \) -- will actually suffice

Will also use

\( \text{isEmpty}(S) \rightarrow \text{boolean} \)
- True if the set \( S \) is empty

\( \text{first}(S) \rightarrow e \)
- returns the least element of \( S \)

\( e \rightarrow S \)
- creates a singleton set from an element

We assume they can both run in \( O(1) \) time.

An ADT with 5 operations!

Union with Split and Join

\[ \text{Union}(S_1, S_2) = \]

if \( \text{isEmpty}(S_1) \) then return \( S_2 \)
else

\((S_L, S_R, f) = \text{Split}(S_2, \text{first}(S_1)) \)

return \( \text{Join}(S_L, \text{Union}(S_2, S_1)) \)

Runtime of Union

Out  \( a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7 \quad a_8 \quad \ldots \)

\( T_{\text{union}} = O(\sum_i \log |a_i| + \sum_j \log |b_j|) \)

Splits joins

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

\( T_{\text{union}} = O(\sum_i \log \lceil n/m + 1 \rceil) = O(m \log ((n+m)/m)) \)
Intersection with Split and Join

\[
\text{Intersect}(S_1, S_2) =
\begin{cases}
  \emptyset & \text{if } \text{isempty}(S_1) \text{ then return } \emptyset \\
  (S_2, S_2, \text{flag}) = \text{Split}(S_2, \text{first}(S_1)) & \text{else}
  \begin{cases}
    \text{return Join(}((\text{first}(S_1)), \text{Intersect}(S_2, S_1)) & \text{if flag then} \\
    \text{return } \text{Intersect}(S_2, S_1) & \text{else}
  \end{cases}
\end{cases}
\]

Efficient Split and Join

Recall that we want: \( T = O(\log |S|) \)

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
- e.g. \((\text{key}, \text{priority}) : (1,23), (4,40), (5,11), (9,35), (12,30)\)

If the priorities are unique, the tree is unique.

Left Spinal Treap

Time to split = length from Start to split location
We will show that this is \( O(\log L) \) in the expected case, where \( L \) is the path length between Start and the split location.
Time to Join is the same
Analysis

\[ P_i = \text{length of path from Start to } i \]
\[ \mathbb{E}[P_i] = p_i \]

\[ A_{ij} = \begin{cases} 
1 & \text{if } x_i \text{ is an ancestor of } x_j \\
0 & \text{otherwise}
\end{cases} \]
\[ \mathbb{E}[A_{ij}] = a_{ij} \]

\[ C_{ilm} = \begin{cases} 
1 & \text{if } x_i \text{ is the common ancestor of } x_l \text{ and } x_m \\
0 & \text{otherwise}
\end{cases} \]
\[ \mathbb{E}[C_{ilm}] = c_{ilm} \]

\[ P_i = \sum_{j=1}^{i} A_{1j} + \sum_{i=1}^{\infty} (A_{il} - C_{ilm}) \]

Analysis Continued

\[ \sum_{i=1}^{\infty} \frac{1}{i} = \sum_{i=1}^{\infty} \frac{1}{i(i-1)+1} = \sum_{i=1}^{\infty} \frac{1}{i} < 1 + \ln i \] (harmonic number \( H_i \))

Can similarly show that:
\[ \sum_{i=1}^{\infty} (a_{ij} - c_{ilm}) = O(\log i) \]

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

\[ \mathbb{E}[P_i] = p_i = \sum_{j=1}^{i} a_{ij} + \sum_{i=1}^{\infty} (a_{il} - c_{ilm}) \]

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 "Posting Lists"

We showed how to take Unions and Intersections, but Treaps are not very space efficient.

Idea: if priorities are in the range \([0,1]\) then any node with priority \( \times 1 - \alpha \) is stored compressed.

\( \alpha \) represents fraction of uncompressed nodes.
Case Study: AltaVista

How AltaVista implements indexing and searching, or at least how they did in 1998.
Based on a talk by Broder and Henzinger from AltaVista. Henzinger is now at Google, Broder is at IBM.
- The index (posting lists)
- The lexicon
- Query merging (or, and, andnot queries)

The size of their whole index is about 30% the size of the original documents it encodes.

AltaVista: the index

All documents are concatenated together into one sequence of terms (stop words removed).
- This allows proximity queries
- Other companies do not do this, but do proximity tests in a postprocessing phase
- Tokens separate documents
Posting lists contain pointers to individual terms in the single "concatenated" document.
- Difference encoded
Use Front Coding for the Lexicon

AltaVista: the lexicon

The Lexicon is front coded.
- Allows prefix queries, but requires prefix to be at least 3 characters (otherwise too many hits)

AltaVista: query merging

Support expressions on terms involving: AND, OR, ANDNOT and NEAR
Implement posting list with an abstract data type called an "Index Stream Reader" (ISR).
Supports the following operations:
- loc () : current location in ISR
- next () : advance to the next location
- seek (k) : advance to first location past k
AltaVista: query merging (cont.)

Queries are decomposed into the following operations:

- **Create**: term → ISR  
  ISR for the term
- **Or**: ISR * ISR → ISR  
  Union
- **And**: ISR * ISR → ISR  
  Intersection
- **AndNot**: ISR * ISR → ISR  
  Set difference
- **Near**: ISR * ISR → ISR  
  Intersection, almost

Note that all can be implemented with our Treap Data structure.

I believe (from private conversations) that they use a two level hierarchy that approximates the advantages of balanced trees (e.g. treaps).