CSCI 5417
Information Retrieval Systems

Jim Martin

Lecture 3
8/30/2010

Today 8/30

- Review
  - Conjunctive queries (intersect)
  - Dictionary contents
  - Phrasal queries
- Tolerant query handling
  - Wildcards
  - Spelling correction

Index: Dictionary and Postings

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

→

Term	N docs	Coll freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Boolean AND: Intersection (1)

```
INTERSECT(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
        then ADD(answer, docID(p1))
             p1 ← next(p1)
             p2 ← next(p2)
        else if docID(p1) < docID(p2)
               then p1 ← next(p1)
               else p2 ← next(p2)
  return answer
```

Boolean AND: Intersection (2)

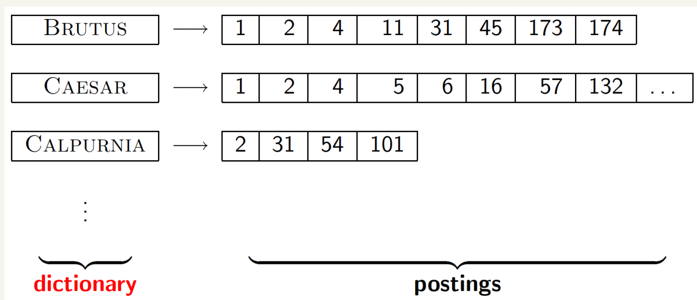
```
INTERSECT(⟨t_1, ..., t_n⟩)
  terms ← SORTBYINCREASINGFREQUENCY(⟨t_1, ..., t_n⟩)
  result ← postings(first(terms))
  terms ← rest(terms)
  while terms ≠ NIL and result ≠ NIL
  do result ← INTERSECT(result, postings(first(terms)))
      terms ← rest(terms)
  return result
```


Review: Dictionary

- What goes into creating the terms that make it into the dictionary?
  - Tokenization
  - Case folding
  - Stemming
  - Stop-listing
  - Normalization
  - Dealing with numbers (and number-like entities)
  - Complex morphology

Dictionary

- The dictionary data structure stores the term vocabulary, document frequency, and pointers to each postings list. In what kind of data structure?



A Naïve Dictionary

- An array of structs?

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
...	...	...
zulu	221	⟶

char[20]   int   postings *

20 bytes   4/8 bytes   4/8 bytes

- How do we quickly look up elements at query time?

Dictionary Data Structures

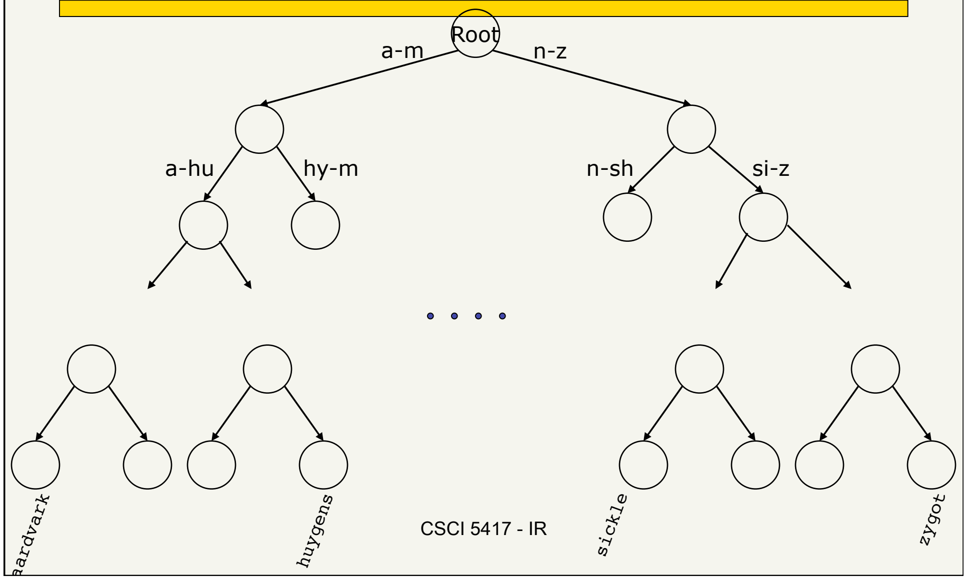
- Two main choices:
  - Hash tables
  - Trees
- Some IR systems use hashes, some trees. Choice depends on the application details.

Hashes

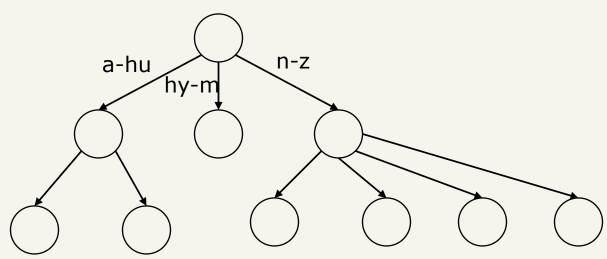
- Each vocabulary term is hashed to an integer
  - I assume you've seen hashtables before
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally rehash *everything*

Sec. 3.1

Binary Tree Approach



Tree: B-tree



- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a, b* are appropriate natural numbers, e.g., [2,4].

Trees

- Simplest approach: binary trees
- More typical : B-trees
- Trees require a standard ordering of characters and hence strings … but we have that
- Pros:
  - Facilitates prefix processing (terms starting with *hyp*)
    - Google's "search as you type"
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

Back to Query Processing

- Users are so demanding...
- In addition to phrasal queries, they like to
  - Use wild-card queries
  - Misspell stuff
- So we better see what we can do about those things

Phrasal queries

- Want to handle queries such as
  - "*Colorado Buffaloes*" – as a phrase
  - This concept is popular with users; about 10% of ad hoc web queries are phrasal queries
- Postings that consist of document lists alone are not sufficient to handle phrasal queries
- Two general approaches
  - Word N-gram indexing
  - Positional indexing

Positional Indexing

- Change the content of the postings
- Store, for each ***term***, entries of the form:

  <number of docs containing ***term***;
  doc1: position1, position2 … ;
  doc2: position1, position2 … ;
  etc.>

Positional index example

<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

Processing a phrase query

- Extract postings for each distinct term: ***to, be, or, not.***
- Merge their *doc:position* lists to enumerate all positions with "***to be or not to be***".
  - ***to****:*
    - *2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...
  - ***be****:*
    - *1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...
- Same general method for proximity searches ("near" operator).

Rules of thumb

- Positional index size 35–50% of volume of original text
- Caveat: all of this holds for “English-like” languages

Wild Card Queries

- Two flavors
  - Word-based
    - Caribb*
  - Phrasal
    - “Pirates * Caribbean”
- General approach
  - Spawn a new set of queries from the original query
    - Basically a dictionary operation
- Run each of those queries in a not totally stupid way

Simple Single Wild-card Queries: *

- Single instance of a *
  - * means an string of length 0 or more
    - This is not Kleene *.
- ***mon*:*** find all docs containing any word beginning "mon".
- Using trees to implement the dictionary gives you prefixes
- ****mon:*** find words ending in "mon"
  - Maintain a backwards index

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- For example, consider the query

  mon* AND octob*

  This results in the execution of many Boolean *AND* queries.

Arbitrary Wildcards

- How can we handle *'s in the middle of query term?
- The solution: transform every possible wild-card query so that the *'s occur at the end
- This motivates the *Permuterm Index*
  - The dictionary/tree scheme remains the same; but we populate the dictionary with extra (special) terms

Permuterm Index

- For the real term ***hello*** create entries under:
  - ***hello$, ello$h, llohe, lohel, o$hell***

  where $ is a special symbol.
- Example

Query = ***hel*o***
Add the $
= ***hel*o$***
Rotate * to the back

Lookup ***o$hel****

Permuterm index

- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell***

  where **$** is a special symbol.
- Queries:
  - **X** lookup on **X$**
  - **X*** lookup on $**X***
  - ***X** lookup on **X$***
  - ***X*** lookup on **X***
  - **X*Y** lookup on **Y$X***

Permuterm query processing

- Rotate query wild-card to the right
- Now use indexed lookup as before.
- *Permuterm problem: ≈ quadruples lexicon size*

Empirical observation for English.

Notice...

- For every new type of query that we'd like to provide to users, a change to the index is required
  - Either to the postings
    - As in phrases
  - Or to the dictionary
    - As in wildcards
- And normally that change means that the index gets bigger
  - That may have an impact on memory management issues

Programming Assignment 1

Questions?

Spelling Correction

- Two primary uses
  - Correcting document(s) being indexed
  - Retrieve matching documents when query contains a spelling error
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words e.g., ***from*** → ***form***
  - Context-sensitive
    - Look at surrounding words, e.g., ***I flew form Heathrow to Narita.***

Document correction

- Primarily for OCR'ed documents
  - Spelling correction algorithms must be tuned for this case
    - Think of Google Books
- The index (dictionary) should contain fewer OCR-induced misspellings
- Can use domain-specific knowledge
  - OCR confuses O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).

Google Books

- The last library...
  - Scan and make available via the Web all the worlds books
    - Around 10M books scanned thus far
    - How many books are there anyway?

Google Books

- Scanning
  - Getting the words right
- Metadata
  - Getting authors, dates, number of pages, copyrights, publisher(s), etc.
    - Some is gotten from content providers (libraries, publishers)
- Use of CAPTCHA for OCR difficulties

Query correction

- Our principal focus here
  - Examples like the query ***Alanis Morisett***
- We can either
  - Retrieve using that spelling
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean … ?*
    - This requires an interactive session

Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Some basic choices for this
  - A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - The lexicon derived from the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the misspellings)

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What's "closest"?
- Several alternatives
  - Edit distance
  - Weighted edit distance
  - Bayesian models
  - Character *n*-gram overlap

Edit distance

- Given two strings S_1 and S_2, the minimum number of basic operations to covert one to the other
- Basic operations are typically character-level
  - Insert
  - Delete
  - Replace
- E.g., the edit distance from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming via minimum edit distance

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture keyboard errors, e.g. ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - (Same ideas usable for OCR, but with different weights)
- Require weight matrix as input
- Modify dynamic programming to handle weights (Viterbi)

Using edit distances

- Given query, first enumerate all dictionary terms within a preset (weighted) edit distance
- Then look up enumerated dictionary terms in the term-document inverted index

Edit distance to all dictionary terms?

- Given a (misspelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
- How do we cut the set of candidate dictionary terms?
  - Heuristics
    - Assume first letter(s) is correct
    - Character *n*-gram overlap

Context-sensitive spell correction

- Text: ***I flew from Heathrow to Narita.***
- Consider the phrase query ***"flew form Heathrow"***
- We'd like to respond

Did you mean "***flew from Heathrow***"?

because ***no*** docs matched the query phrase.

Context-sensitive correction

- First idea: retrieve dictionary terms close (in weighted edit distance) to each query term
- Now try all possible resulting phrases with one word “fixed” at a time
  - ***flew from heathrow***
  - ***fled form heathrow***
  - ***flea form heathrow***
  - ***etc.***
- Suggest the alternative that has lots of hits
- Suggest the alternative that matches previous queries?

General issue in spell correction

- Will enumerate multiple alternatives for “Did you mean”
- Need to figure out which one (or small number) to present to the user
- Use heuristics
  - The alternative hitting most docs
  - Query log analysis + tweaking
    - For especially popular, topical queries
  - Language modeling

Next Time

- On to Chapter 4
  - Back to indexing