CS3245

Information Retrieval

4

Lecture 4: Dictionaries and Tolerant Retrieval

Last Time: Terms and Postings Details

- Skip pointers
  - Encoding a tree-like structure in a postings list
- Biword indexes for phrases
- Positional indexes for phrases/proximity queries

- The term/token distinction
  - Terms are (modified) tokens selected for the dictionary
- Tokenization problems
  - Hyphens, apostrophes, spaces, compounds
  - Language-specific problems
- From token to terms
  - Normalization, lemmatization/stemming

Today: the dictionary and tolerant retrieval

- Dictionary data structures
- “Tolerant” retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?

BRUTUS	→	1	2	4	11	31	45	173	174	
CAESAR	→	1	2	4	5	6	16	57	132	...
CALPURNIA	→	2	31	54	101					
⋮										

dictionary

postings

A naïve dictionary

- An array of struct:

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

`char[20]` `int` `Postings Pointer`

20 bytes 4/8 bytes 4/8 bytes

Quick Q: What's wrong with using this data structure?

A naïve dictionary

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

char[20]	int	Postings Pointer
20 bytes	4/8 bytes	4/8 bytes

Words can only be 20 chars long. Waste of space for some words, not enough for others.

- How do we store a dictionary in memory efficiently?

Most important: Slow to access, linear scan needed!

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

Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
- Some IR systems use hashes, some trees

To think about: what issues influence the choice between these two data structures? (Hint: see IIR)

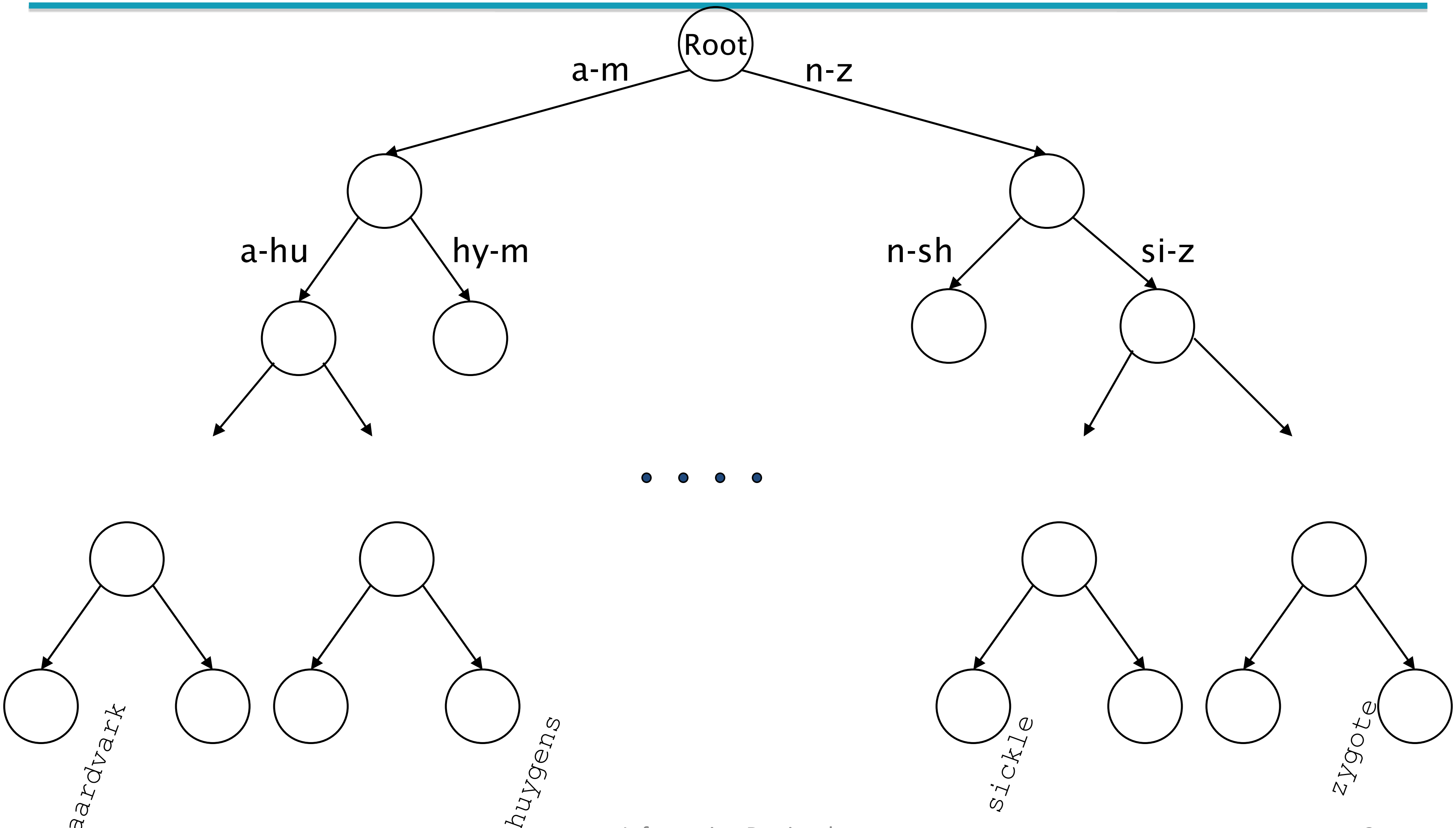
Hash Table

Each vocabulary term is hashed to an integer

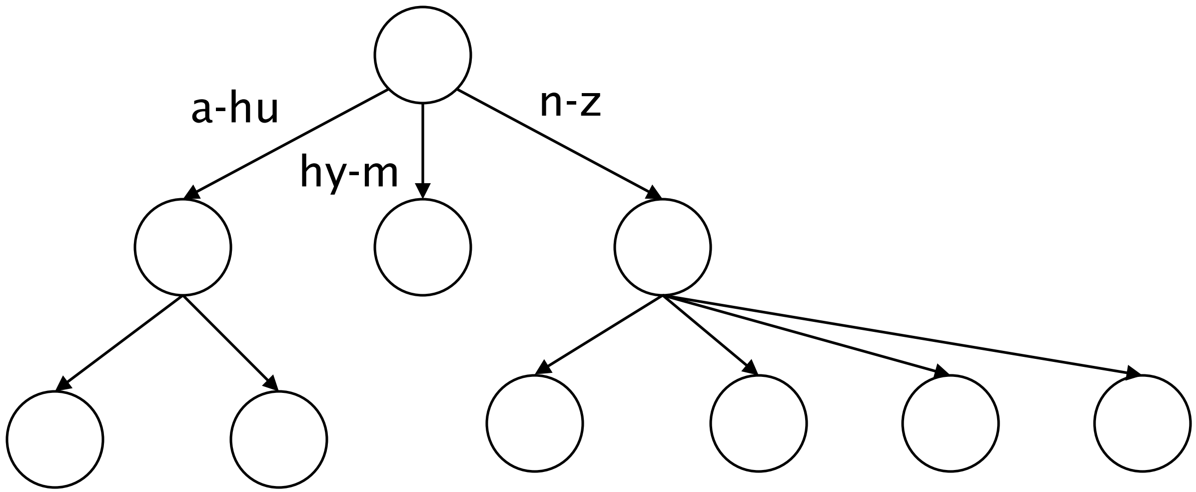
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Not very tolerant!

Tree: binary tree



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: binary tree
- More common: B-trees
- Trees require a standard ordering of characters and hence strings … but we have one: lexicographical ordering
- Pros:
  - Solves the prefix problem (e.g., terms starting with *"hyp"*)
- Cons:
  - Slower: O(log M) [and this requires a *balanced* tree]
  - Rebalancing binary trees is expensive
    - B-trees mitigate the rebalancing problem

WILDCARD QUERIES

Wildcard queries: *

- ***mon*:*** find all docs containing any word beginning "mon".

Quick Q1: why would someone use this feature?

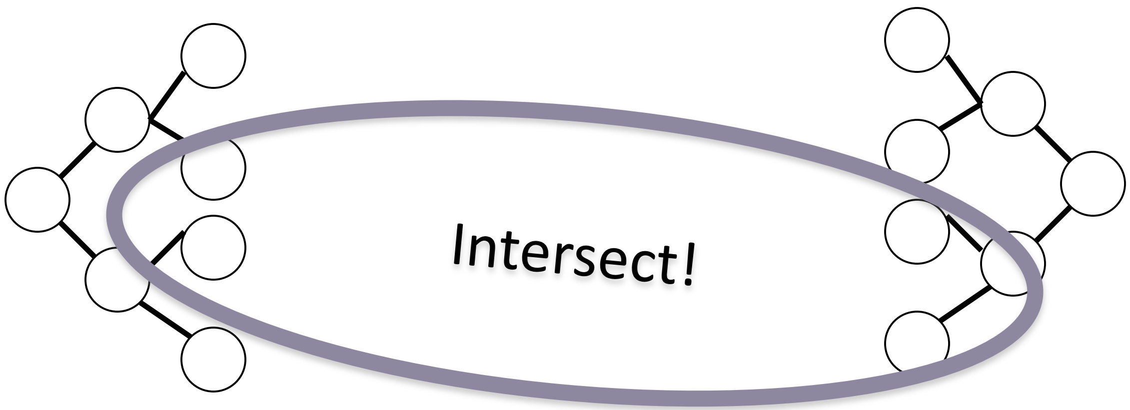
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo***
- ****mon:*** find words ending in "mon": need help!
  - Maintain an additional B-tree for terms reversed

Can retrieve all words in range: ***nom ≤ w < non***.

Quick Q2: from this, how can we enumerate all terms meeting the wildcard query ***pro*cent*** ?

Intersection, redux

Answer: Use the forward part for "pro*", and the backward part for "*cent", then intersect them.



Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wildcard query.
- We still have to look up the postings for each enumerated term ➔ still expensive

- E.g., consider the query:

  se*ate *AND* ***fil*er***

  This may result in the execution of many Boolean *AND* queries.

B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
  - ***co*tion***
- We could look up ***co**** AND ****tion*** in a B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the *'s always occur at the end
- This gives rise to the **Permuterm** Index.

Permuterm index

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

  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***

Query = hel*o
X=hel, Y=o
Lookup o$hel*

Not so quick Q:
What about X*Y*Z?

Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before

- *Permuterm problem: lexicon size blows up, proportional to average word length*

Is there any other solution?

Bigram (*k*-gram) indexes

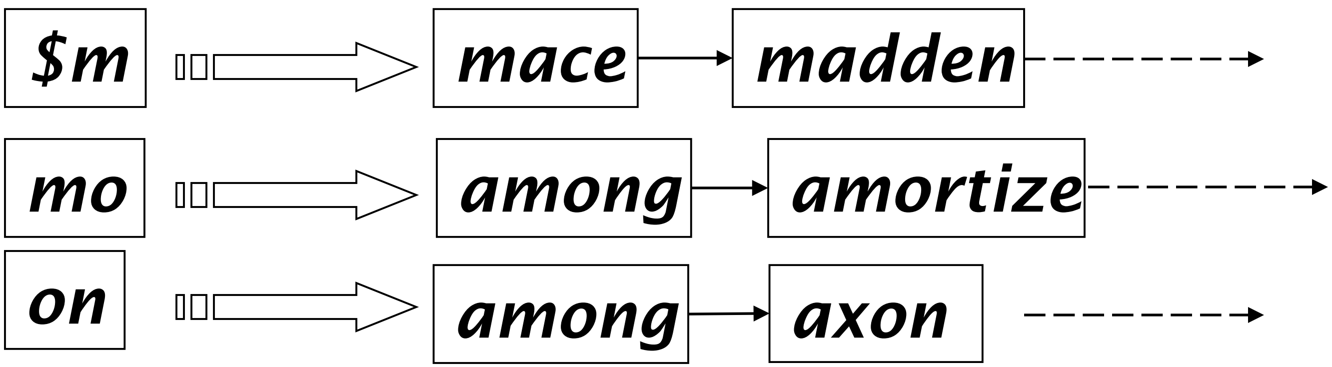
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.,* from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

a,ap,pr,ri,il,l,i,is,s,t,th,he,e,$c,cr,ru, ue,el,le,es,st,t$,$m,mo,on,nt,h$

  - As before "$" is a special word boundary symbol
- Maintain a <u>*second*</u> inverted index <u>*from bigrams to dictionary terms*</u> that match each bigram.

Bigram index example

- The *k*-gram index finds *terms* based on a query consisting of *k*-grams (here *k*=2).



Processing wildcards

- Query ***mon**** can now be run as
  - ***$m** AND **mo** AND **on***
- Gets terms that match AND version of our wildcard query.

- Oops! We also included ***moon***, a false positive!
- Must post-filter these terms against query.
- Surviving enumerated terms are then looked up in the term-document inverted index.
- Fast, space efficient (compared to permuterm).

Processing wildcard queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wildcards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage laziness, people will respond!

Search

Type your search terms, use ‘*’ if you need to.
E.g., Alex* will match Alexander.

Which web search engines allow wildcard queries?

SPELLING CORRECTION

Spellling corektion

- Two principal uses:
    1. Correcting document(s) being indexed
    2. Correcting user queries to retrieve “right” answers

- 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 <u>form</u> Heathrow to Narita.***

Document correction

- Especially needed for OCR’ed documents
  - Correction algorithms are tuned for common errors: `rn/m`
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).
- But also: web pages and even printed material have typos
- Goal: the dictionary contains fewer misspellings
- But often we don’t change the documents but aim to fix the query-document mapping

Query misspellings

- Our principal focus here
  - E.g., the query ***Britiny Speares***
- We can either
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *"Did you mean … ?"*

Isolated word correction

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

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- How do we define "closest"?
- We'll study several alternatives
  1. Edit distance (Levenshtein distance)
  2. Weighted edit distance
  3. *n*gram overlap

1. Edit distance

- Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
  - Fundamentally related to the longest common subsequence (LCS) problem you may already know
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from ***dof*** to ***dog*** is 1
  - From ***cat*** to ***act*** is 2. (Just 1 with transpose)
  - from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming

Dynamic Programming

Not dynamic and *not* programming

- Build up solutions of "simpler" instances from small to large
  - Save results of solutions of "simpler" instances
  - Use those solutions to solve larger problems
- Useful when problem can be solved using solution of two or more instances that are only slightly simpler than original instances

Computing Edit Distance

Let's diagram this as an array, with

S_1 (PAT) on the x-axis,

S_2 (APT) on the y-axis.

Store edit distance between substrings $S_{1(1,i)}$ and $S_{2(1,j)}$ at entry i,j

S_2 \ S_1	_	**P**	**A**	**T**
_	0	1	2	3
A	1	1	1	2
P	2	1	2	2
T	3	2	2	2

Possible moves:

- Match or substitute
- Insert: Insert a character in S_1
- Delete: Delete a character in S_2

$$E(i, j) = \min\{ E(i, j-1) + 1,\ E(i-1, j) + 1,\ E(i-1, j-1) + \mathbf{m}\}$$

where $\mathbf{m}$ = 1 if $P_i \neq T_j$, 0 otherwise

Blanks on slides, you may want to fill in

Practice your edit distance

	_	C	H	I	C	K	E	N
_	0	1	2	3	4	5	6	7
C	1							
H	2							
E	3							
E	4							
K	5							
Y	6							

Blanks on slides, you may want to fill in

Practice your edit distance

	_	C	H	I	C	K	E	N
_	0	1	2	3	4	5	6	7
C	1	0	1	2	3	4	5	6
H	2	1	0	1	2	3	4	5
E	3	2	1	1	?			
E								
K								
Y								

2. Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or 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***
  - This may be formulated as a probability model
- Requires a weighted matrix as input
- Modify dynamic programming to handle weights

Using edit distances

- Given query, first enumerate all character sequences within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with list of “correct” words
- Show terms you found to user as suggestions
- Alternatively,
  - We can look up all possible corrections in our inverted index and return all docs … slow
  - We can run with a single most likely correction
- The alternatives disempower the user, but may save a round of interaction with the user

Edit distance to all dictionary terms?

- Given a (misspelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
  - Alternative?
- How do we cut the set of candidate dictionary terms?
  - One possibility is to use *n*gram overlap for this
  - This can also be used by itself for spelling correction

3. Ngram overlap

- Enumerate all the ngrams in the query string as well as in the lexicon
- Use the ngram index (recall wildcard search) to retrieve all lexicon terms matching any of the query ngrams
- Threshold by number of matching ngrams
  - Variants – weight by keyboard layout, assume initial letter correct, etc.

Arocdnicg to rsceearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer are in the rghit pcale. The rset can be a toatl mses and you can sitll raed it wouthit pobelrm. Tihs is buseace the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

This story is actually an urban legend? No such study was done at Cambridge

Example with trigrams

- Suppose the text is ***november***
  - Trigrams are *nov, ove, vem, emb, mbe, ber*.
- The query is ***december***
  - Trigrams are *dec, ece, cem, emb, mbe, ber*.
- So 3 trigrams overlap (out of 6 in each term)

How can we turn this into a normalized measure of overlap?

One option – Jaccard coefficient

- A commonly-used measure of overlap
- Let *X* and *Y* be two sets; then the J.C. is

$$\left|X \cap Y\right| / \left|X \cup Y\right|$$

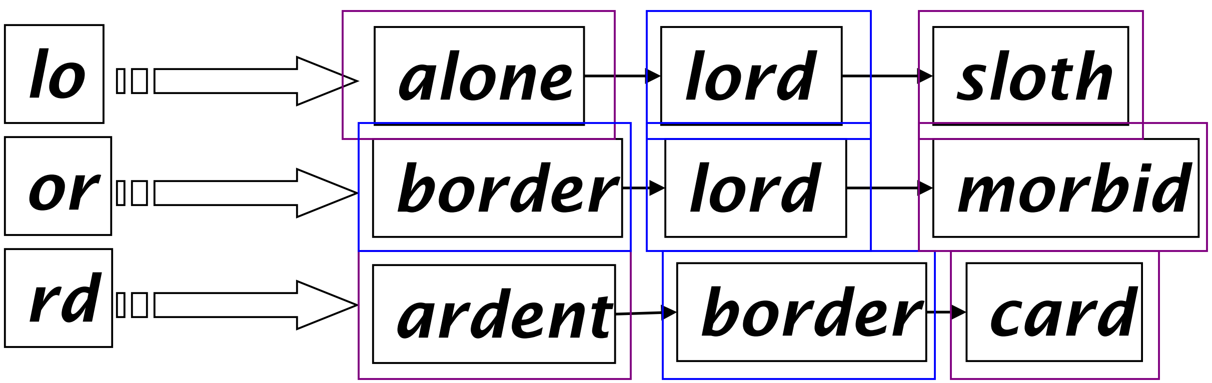
A generally useful overlap measure, even outside of IR

- Equals 1 when *X* and *Y* have the same elements and zero when they are disjoint
- *X* and *Y* don't have to be of the same size
- Always assigns a number between 0 and 1
  - Now threshold to decide if you have a match
  - E.g., if Jaccard > 0.8, declare a match

"coefficient de communauté"

Matching trigrams

- Consider the query ***lord*** – we wish to identify words matching 2 of its 3 bigrams (***lo, or, rd***)



Standard postings "merge" enumerates hits

Adapt this to using Jaccard (or another) measure.

Context-sensitive spelling 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

- Need surrounding context to catch this.
- 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***
- **Hit-based spelling correction:** Suggest the alternative that has lots of hits (in queries or documents)

The correct query *"flew from munich"* has the most hits

The **hit-based paradigm** is applied in many other places too!

Another approach

- Break phrase queries into conjunctions of biwords.
- Look for biwords that need only one term corrected.
- Enumerate phrase matches and … rank them!

More on this later

General issues in spelling correction

- We enumerate multiple alternatives for "Did you mean?" but we need to decide which to present to the user
- Use heuristics
  - The alternative hitting most docs
  - Query log analysis + tweaking
    - For especially popular, topical queries
- Spelling correction is computationally expensive
  - Avoid running routinely on every query?
  - Run only on queries that matched few docs

SOUNDEX

Blanks on slides, you may want to fill in

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for **names**
  - E.g., ***chebyshev*** → ***tchebycheff***
- Invented for the U.S. census

- We'll explore this just in the context of English

To think about: what other languages does it make sense for?

Soundex – typical algorithm

- Turn every token to be indexed into a 4-character reduced form
- Do the same with query terms
- Build and search an index on the reduced forms
  (when the query calls for a Soundex match)

- See Wikipedia's entry: https://en.wikipedia.org/wiki/Soundex

Soundex – typical algorithm

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:
   - B, F, P, V → 1
   - C, G, J, K, Q, S, X, Z → 2
   - D,T → 3
   - L → 4
   - M, N → 5
   - R → 6

Soundex continued

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

Will ***hermann*** generate the same code?

Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, …)

How useful is Soundex?

- Not very – for general IR, spelling correction
- Okay for "high recall" tasks (e.g., Interpol), though biased to names of certain nationalities
  - Sucks for Chinese names: Xin (Pinyin) and Hsin (Wade-Giles) mapped completely different

Now what queries can we process?

- We have
  - Positional inverted index with skip pointers
  - Wildcard index
  - Spelling correction
  - Soundex
- Queries such as

(SPELL(moriset) /3 toron*to) OR *SOUNDEX* ***(chaikofski)***

Summary

- Data Structures for the Dictionary
  - Hash
  - Trees

- Learning to be tolerant
1. Wildcards
   - General Trees
   - Permuterm
   - Ngrams, redux
2. Spelling Correction
   - Edit Distance
   - Ngrams, re-redux
3. Phonetic – Soundex

Resources

- IIR 3, MG 4.2
- Efficient spelling retrieval:
  - K. Kukich. Techniques for automatically correcting words in text. ACM Computing Surveys 24(4), Dec 1992.
  - J. Zobel and P. Dart. Finding approximate matches in large lexicons. Software - practice and experience 25(3), March 1995. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.3856&rep=rep1&type=pdf
  - Mikael Tillenius: Efficient Generation and Ranking of Spelling Error Corrections. Master's thesis at Sweden's Royal Institute of Technology. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1392
- **Nice, easy reading on spelling correction:**
  - Peter Norvig: How to write a spelling corrector

http://norvig.com/spell-correct.html

It's in python!