CS3245

Information Retrieval

4

Lecture 4: Dictionaries and Tolerant Retrieval

Live Q&A
https://pollev.com/jin

Last Time: Postings lists and Choosing terms

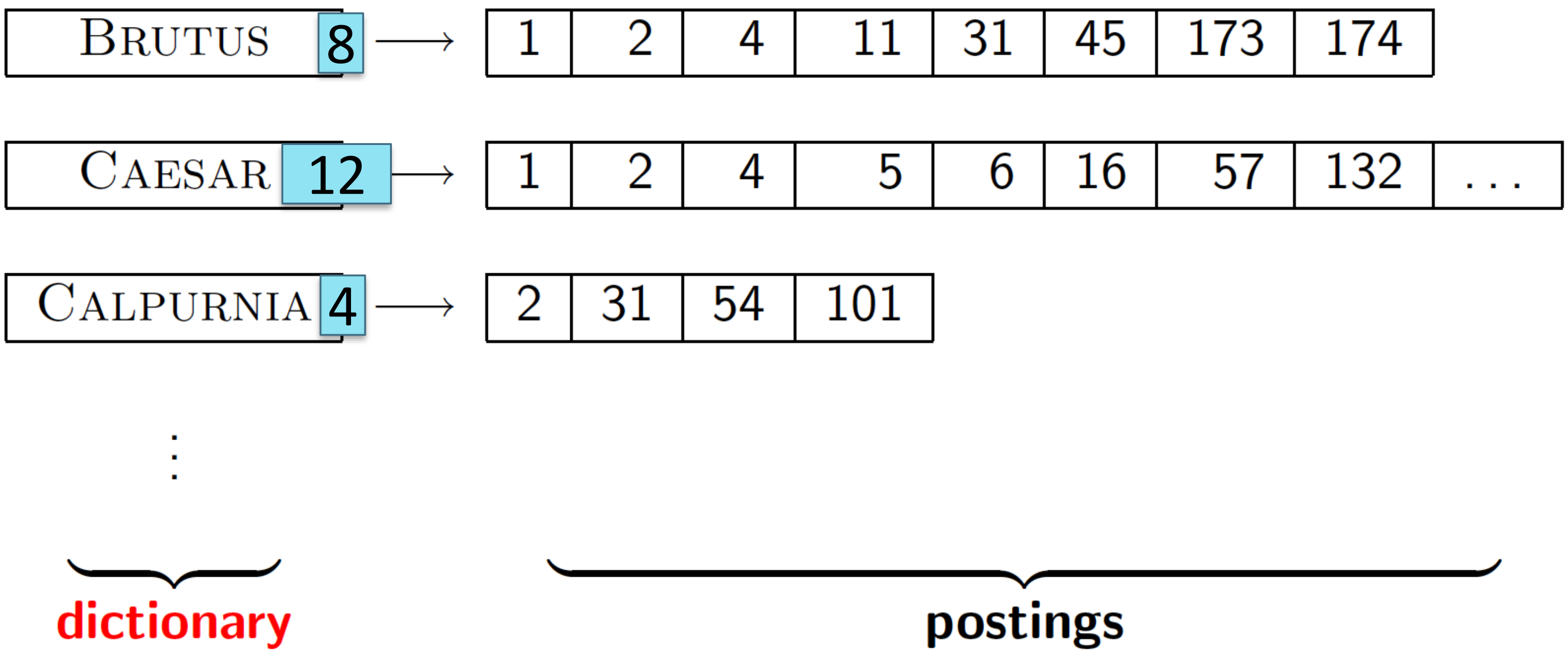
- Faster merging of posting lists
  - Skip pointers
- Handling of phrase and proximity queries
  - Biword indexes for phrase queries
  - Positional indexes for phrase/proximity queries
- Steps in choosing terms for the dictionary
  - Text extraction
  - Granularity of indexing
  - Tokenization
  - Stop word removal
  - Normalization
  - Lemmatization and stemming

Today: Tolerant retrieval

- "Tolerant" retrieval
  - Dictionary
  - Wild-card queries (e.g., cat*)
  - Spelling correction (e.g., Standford University)

Dictionary

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



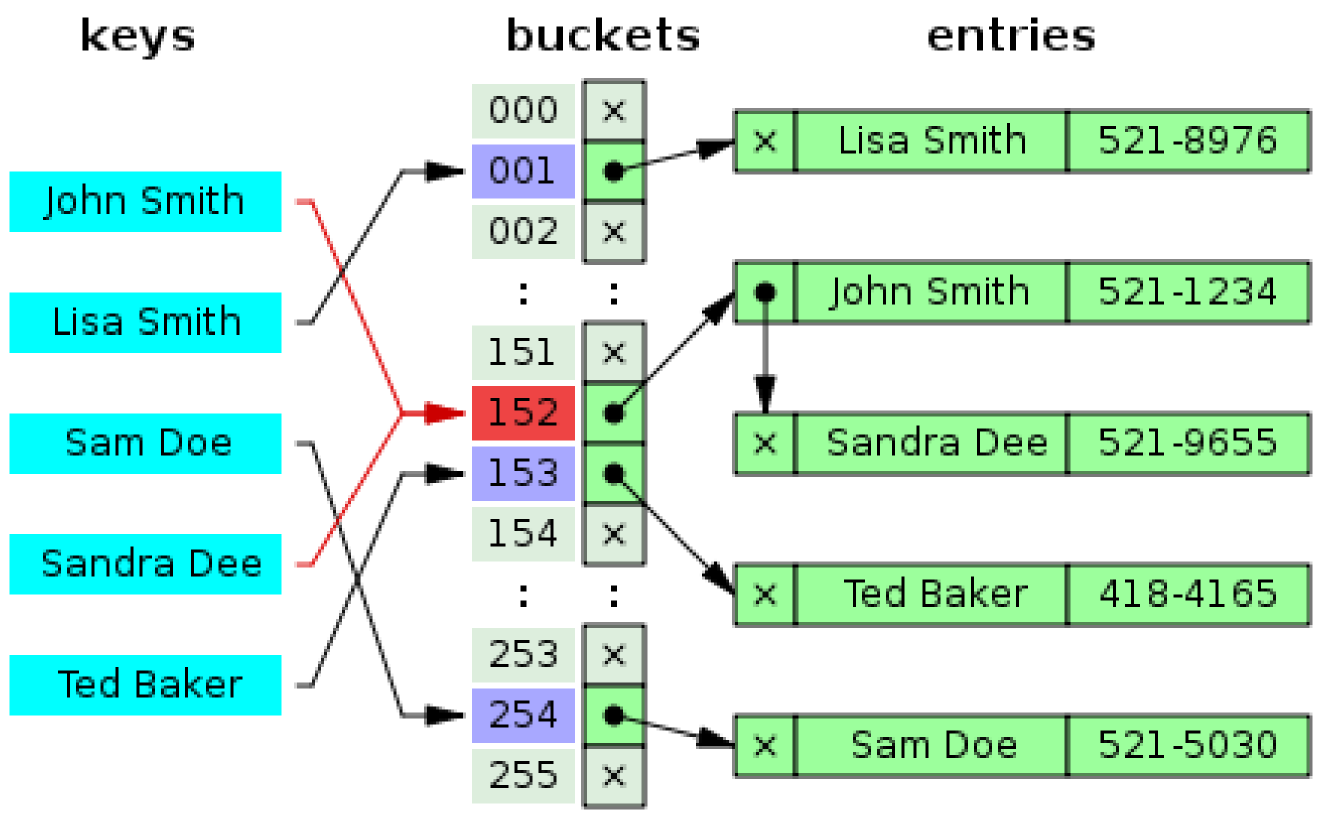
A naïve dictionary

- Storing the entries sequentially in an array:

	term	document frequency	pointer to postings list
dict[0]	a	656,265	⟶
dict[1]	aachen	65	⟶
…	. . .	. . .	. . .
dict[…]	zulu	221	⟶

- Costly to maintain sortedness for fast access
- Lack of support for tolerant retrieval

Main choice 1: Hash Table

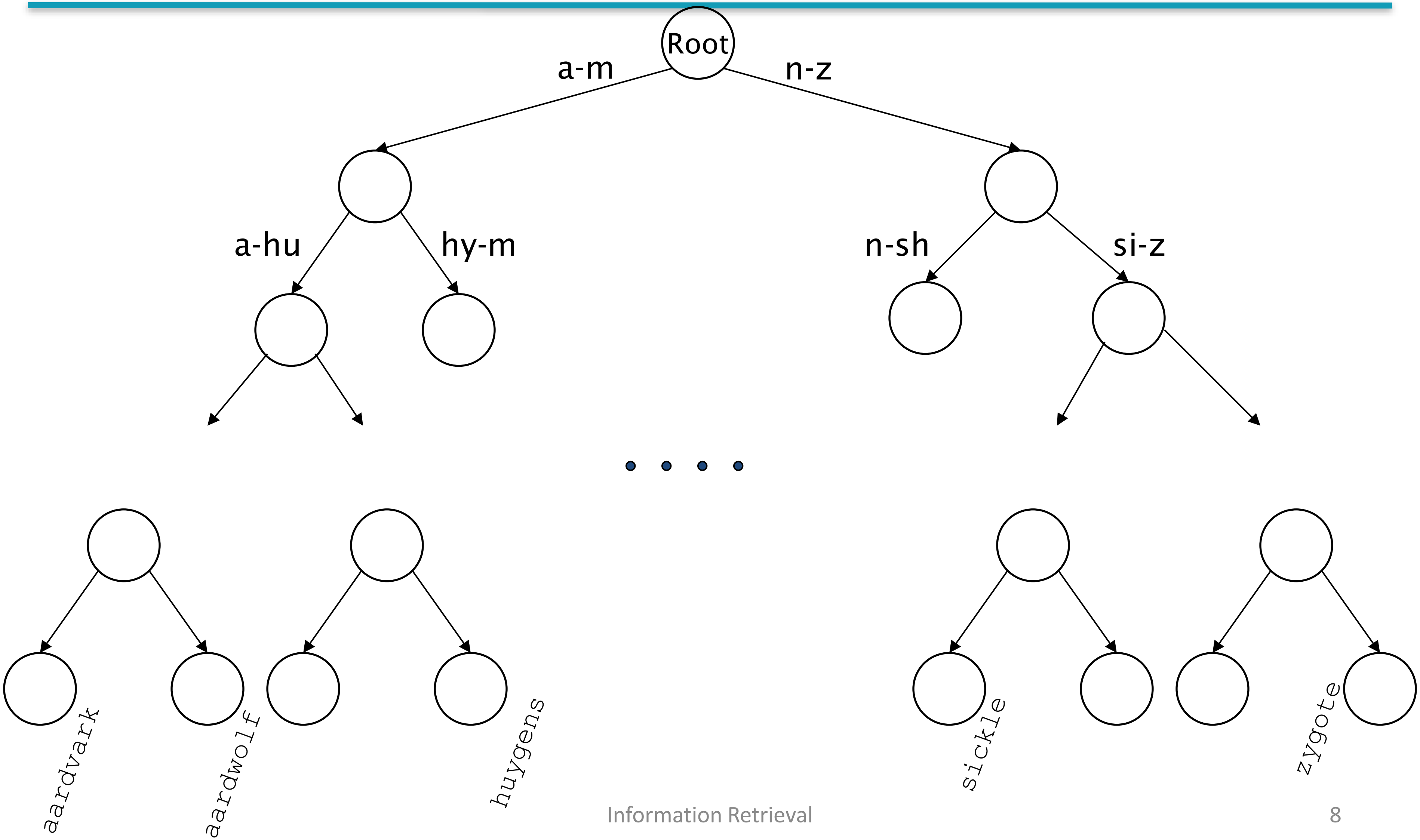


Main choice 1: Hash Table

- Pros:
  - Faster: O(1) for lookup
  - Handles changes well (unless a re-hash is required)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search (e.g., terms starting with *"hyp"*)

Not very tolerant!

Main choice 2: Tree



Main choice 2: Tree

- Pros:
  - Handles changes well (via re-balancing)
  - Solves the prefix problem (e.g., terms starting with "*mon*")
  - Easier to find minor variants:
    - judgment/judgement

More tolerant!

- Cons:
  - Slower (than Hash Table): O(log *M*) on a balanced tree

WILDCARD QUERIES

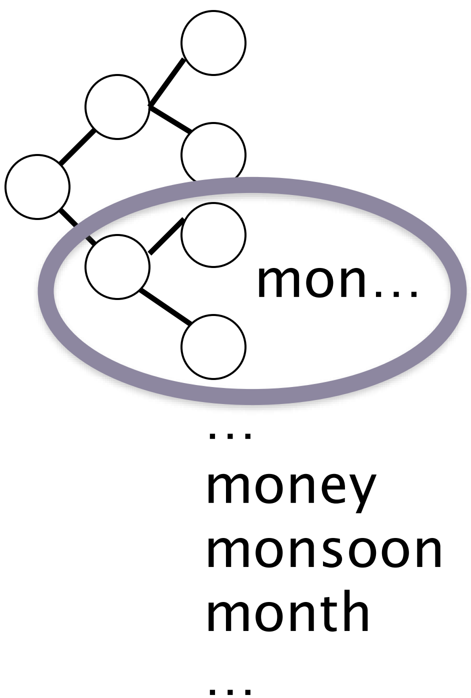
Wildcard queries: *

- * matches with any sequence of letters

- Sample use cases
  - File search based on extension (e.g., *.jpg)
  - Variation in spelling (e.g., col*ur)
  - Single vs plural form (e.g., cat*)
  - …

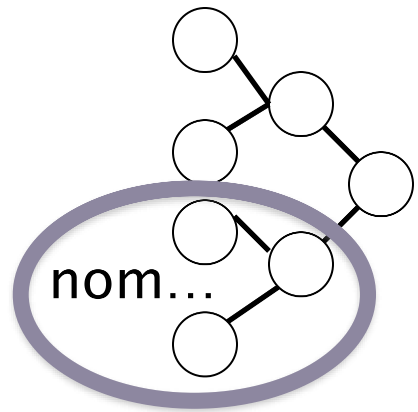
Wildcard queries: *

- ***mon*:*** find docs with words beginning with "mon".
  - Maintain a binary tree for terms
  - Retrieve all words in range: ***mon ≤ w < moo***



Wildcard queries: *

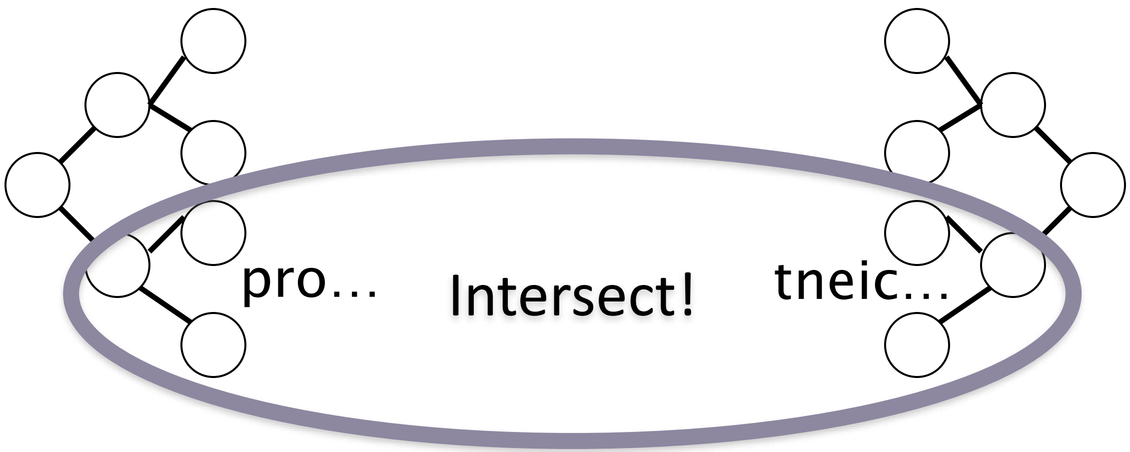
- ***mon:** find docs with words ending in "mon"
  - Maintain an additional tree for terms reversed
  - Retrieve all words in range: ***nom ≤ w < non.***



…
nomel
nomlas
nommoc
…

Handling general wildcard queries

- How about ***pro*cient***?
  - Retrieve possible words for pro* and *cient from the trees and intersect



…
proceed
proficient
profile
…

…
tneiciffe
tneiciffus
tneiciforp
…

Handling general wildcard queries

- General wildcard queries: X*Y
- Look up ***X**** in a normal tree AND ****Y*** in a reverse tree, and then intersect the two term sets
  - Expensive
- The solution: transform wildcard queries into prefix queries (i.e., * occurs at the end)
- This gives rise to the **Permuterm** Index.

Permuterm index

- For the term ***hello***, add an end marker $ and index all rotations:
  - ***hello$, ello$h, llohe, lohel, o$hell*** *and* ***$hello***
- For a wildcard query, add an end marker $ and look up using the rotation with * at the end
  - **X*** lookup on $**X*** ***X** lookup on **X$***
  - **X*Y** lookup on **Y$X*** ***X*** lookup on **X***

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

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

Permuterm index

- Lexicon size blows up, proportional to average word length
  - E.g., A 5-letter word, **hello**, has 6 rotations

Is there any other solution?

Bigram index

- Enumerate all **letter** *bigrams* (sequence of 2 letters) occurring in any term

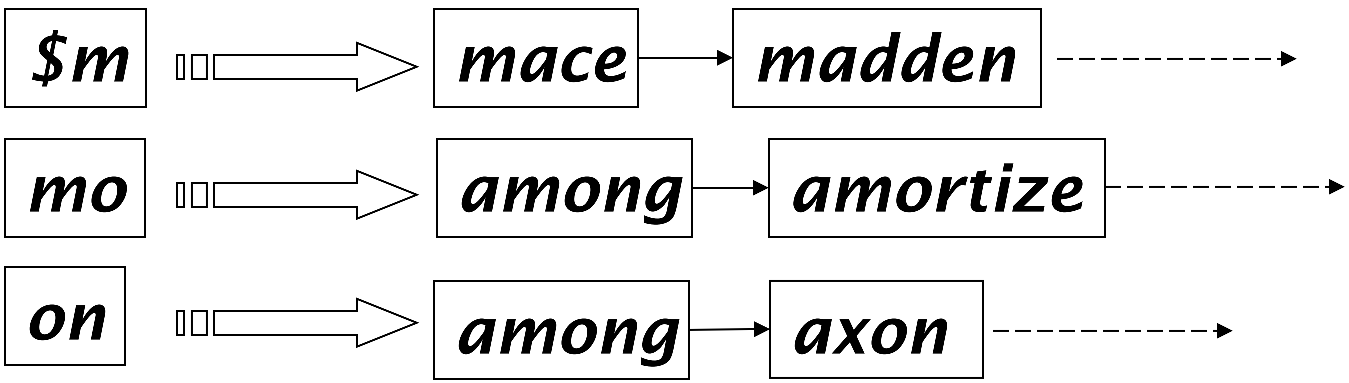
- E.g., From "*among*", we get the 2-grams (*bigrams*)

  a, am, mo, on, ng, g

  - As before "$" is a special word boundary symbol

Bigram index

- Maintain a *second* inverted index *from bigrams to a sorted list of dictionary terms* that contains each bigram.



- Query ***mon**** can now be run as an "AND" Query
  - ***$m** AND **mo** AND **on***
  - Possible matches: **month**, **moon**, …

Bigram index

- Oops! We also included ***moon***, a false positive!
  - It also contains all 3 bigrams **$m, mo, on**
  - 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).
  - Only the original form of a term is stored.
  - TermIDs can be used instead to reduce the space required.
- Can be generalize to k-gram index.

Processing wildcard queries

- After getting the possible terms, we still need to execute a Boolean query for each possible 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

Query misspellings

- Need to correct user queries to retrieve "right" answers
  - E.g., the query ***Ellon Mask***

- We can
  - Return several suggested alternative queries with the correct spelling
    - *"Did you mean … ?"*
  - Retrieve documents indexed by the correct spelling

Spellling corektion

- 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 Narita.***

Fundamental premise

- There is a lexicon of correct spellings.

- Two basic choices for this
  - A standard lexicon, e.g.,
    - 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
  - dof → dog, dock, cat….?
- How do we define "closest"?
- We'll study two alternatives
  1. Edit distance (Levenshtein distance)
  2. ngram overlap

1. Edit distance

- Given two strings S_1 and S_2, the edit distance D (S_1, S_2) is the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace
- E.g., D (**dof** , **dog**) = 1
  - D (**cat**, **act**) = 2.
  - D (**cat**, **dog**) **=** 3.
- Generally found by dynamic programming

Dynamic Programming

Not dynamic and *not* programming

- Build up solutions of "simpler" instances from small to large
  - Compute solutions of "simpler" instances
  - Use these solutions to solve larger problems
  - E.g., Fibonacci numbers

Fib(1)	Fib(2)	Fib(3)	Fib(4)	Fib(5)
1	1	1+1=2	1+2=3	2+3=5

- Useful when problem can be solved using solution of two or more (slightly) simpler problems.

Computing Edit Distance

- Let's try to compute the edit distance between S_1 = **PAT** and S_2 = **APT** using this array E, where
  - E (i, j) = the distance between S_1 (up to the i-th character) and S_2 (up to the j-th character)
  - "_" denotes an empty string

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

- E (0, 0) = D (_, _)
- E (1, 2) = D (P, AP)
- E (3, 3) = D (PAT, APT)

Computing Edit Distance

- E.g., base cases
  - D (_, _) = D (0, 0) = 0
  - D (P, _) = D (1, 0) = 1
  - D (_, A) = D (0, 1) = 1

	i	0	1	2	3
j	S_1 S_2	_	**P**	**A**	**T**
0	_	0	1		
1	**A**	1			
2	**P**				
3	**T**				

Computing Edit Distance

- E.g., recursive cases
  - D (PA, AP) = ??
- What are the smaller problems?
  - If we know D (PA, A), the final distance is D (PA, A) + 1 since we need **one insertion** to add P to the second string.
  - If we know D (P, AP), the final distance is D (P, AP) + 1 since we need **one insertion** to add A to the first string.
  - If we know D (P, A), the final distance is D (P, A) + 1 since **inserting A to both strings** does not change the distance and we need to **replace the A in the second string with P**.
- What is the minimal distance?

Computing Edit Distance

D(PA, AP) @ E (2, 2) = min {
  D(PA, A) @ E(2, 1) + 1,
  D(P, AP) @ E(1, 2) + 1,
  D(P, A) @ E(1, 1) + 1
} = 2

j \ i	S_2 \ S_1	0: _	1: P	2: A	3: T
0	_	0	1	2	
1	A	1	1	1	
2	P	2	1	2	
3	T				

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

where **m** = **1** if $P_i \neq T_j$, **0** otherwise

Computing Edit Distance

- E.g., recursive cases
  - D (PAT, APT) = ??
- What are the smaller problems?
  - If we know D (PAT, AP), the final distance is D (PAT, AP) + 1 since we need **one insertion** to add T to the end of **AP**.
  - If we know D (PA, APT), the final distance is D (PA, APT) + 1 since we need **one insertion** to add T to the end of **PA**.
  - If we know D (PA, AP), the final distance is still D (PA, AP) since **inserting T to both PA and AP** does not change the distance.
- What is the minimal distance?

Computing Edit Distance

D(PAT, APT) @ E (3, 3) = min {

D(PAT, AP) @ E(3, 2) + 1,

D(PA, APT) @ E(2, 3) + 1,

D(PA, AP) @ E(2, 2) + 0

} = 2

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

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

where **m** = **1** if $P_i \neq T_j$, **0** otherwise

Edit distance to all dictionary terms?

- Given a (misspelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
  - Alternative: Consider everything up to distance 1 or 2.

- 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

2. Ngram overlap

- Enumerate all the ngrams in the query string as well as in the lexicon
  - Query term: **lord** → Bigrams: {**lo**, **or**, rd}
  - Lexicon term: **lore** → Bigrams {**lo**, **or**, re}
- Count the overlaps between a pair of terms
  - 1 between lord and alone
  - 2 between lord and lore
  - 3 between lord and overlord

This favors longer terms by nature, why?

- Threshold to decide if you have a match
  - E.g., if count >= 2, declare a match

A normalized option – Jaccard coefficient

- Let *X* and *Y* be two sets; then the J.C. is

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

A generally useful overlap measure, even outside of IR

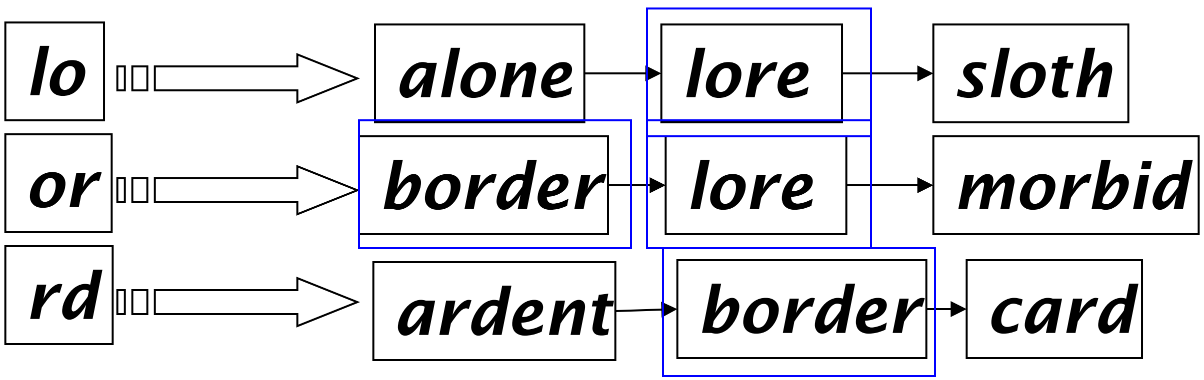
  - Equals 1 when *X* and *Y* have the same elements and 0 when they are disjoint
  - Does not favor longer terms.
  - E.g., JC(lord, lore) = 2/4
    JC(lord, overlord) = 3/7

"*coefficient de communauté*"

- Threshold to decide if you have a match
  - E.g., if count >= 2 AND Jaccard >= 0.5, declare a match

Matching bigrams

- Maintain a letter bigram index!
- Identify words with at least 2 overlaps (and Jaccard >= 0.5) by merging.



Standard postings "merge" enumerates terms with multiple overlaps

Context-sensitive correction

- **Query**: flew form Narita
- Need context to correct "form" to "from"

- Retrieve dictionary terms close (e.g., in edit distance) to each query term

- Enumerate all possible resulting phrases with one word "corrected" at a time
  - *flew **from** Narita*
  - ***fled** form Narita*
  - *flew form **Arita***

Which one to pick?

Context-sensitive correction

- Decide which ones to present using heuristics
  - **Hit-based spelling correction**
    - The correction with most hits

- E.g., *flew* ***from*** *Narita* (100,000 hits) ← pick this!
  fled *form Narita* (200 hits)
  flew form ***Arita*** (500 hits)

General issues in spelling correction

- Confirm with the user vs. search automatically (e.g., with the most possible correction)
  - Disempowerment or effort saved?
- High computational cost
  - Avoid running routinely on every query?
  - Run only on queries that matched few docs

Now what queries can we process?

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

  SPELL(moriset) /3 toron*to

Summary

- Learning to be tolerant
  - Dictionary
    - Hashtable
    - Tree
  - Wildcards
    - Permuterm
    - Ngrams, redux
  - Spelling correction
    - Edit Distance
    - Ngrams, re-redux

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!