Information Retrieval

Weike Pan

The slides are **adapted from those provided by Prof. Hinrich Schütze** at University of Munich (http://www.cis.lmu.de/~hs/teach/14s/ir/).

Chapter 3 Dictionaries and tolerant retrieval

- 3.1 Search structures for dictionaries
- 3.2 Wildcard queries
- 3.3 Spelling correction
- 3.4 Phonetic correction
- 3.5 References and further reading

Outline

- 3.1 Search structures for dictionaries
- 3.2 Wildcard queries
- 3.3 Spelling correction
- 3.4 Phonetic correction
- 3.5 References and further reading

3.1 Search structures for dictionaries

Dictionaries

- Dictionary: the data structure for storing the term vocabulary

3.1 Search structures for dictionaries

Dictionary as an array of fixed-width entries (1/2)

- For each term, we need to store a couple of items:
  - document frequency
  - pointer to postings list
  - ...
- Assumptions
  - we can store this information in a fixed-length entry
  - we store these entries in an array

3.1 Search structures for dictionaries

Dictionary as an array of fixed-width entries (2/2)

	term	document frequency	pointer to postings list
	a	656,265	$\longrightarrow$
	aachen	65	$\longrightarrow$
	...	...	...
	zulu	221	$\longrightarrow$
space needed:	20 bytes	4 bytes	4 bytes

- How do we **look up** a query term in this array?
  - That is: which data structure do we use to locate the entry in the array where the query term is stored?

3.1 Search structures for dictionaries

Data structures for looking up a term

- Two main classes of data structures: hashes and trees
- Some IR systems use hashes and some use trees.

- Hashes vs. trees:
  - Is there a fixed number of terms or will it keep growing?
  - What are the relative frequencies with which various keys will be accessed?
  - How many terms are we likely to have?

3.1 Search structures for dictionaries

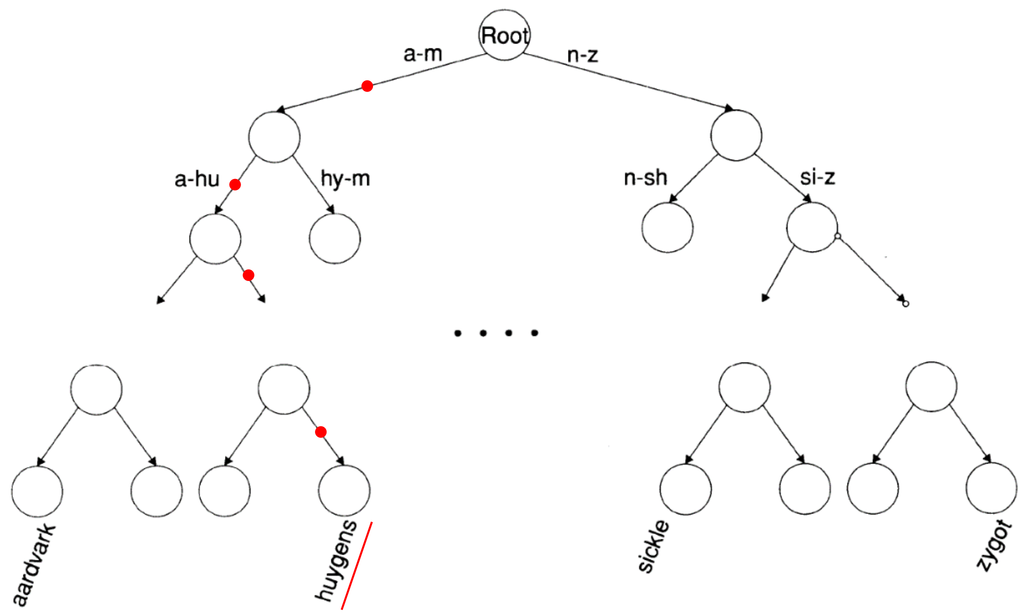
Hashes

- Each vocabulary term is hashed into an integer, i.e., its row number in the array
- At query time: hash a query term and locate an entry in the fixed-width array
- **Pros (优点)**: Lookup in a hash is faster than lookup in a tree. The lookup time is a constant.
- **Cons (缺点)**:
  - no way to find minor variants
  - no prefix search (e.g., all terms starting with automat)
  - need to rehash everything periodically if the vocabulary keeps growing

3.1 Search structures for dictionaries

Trees

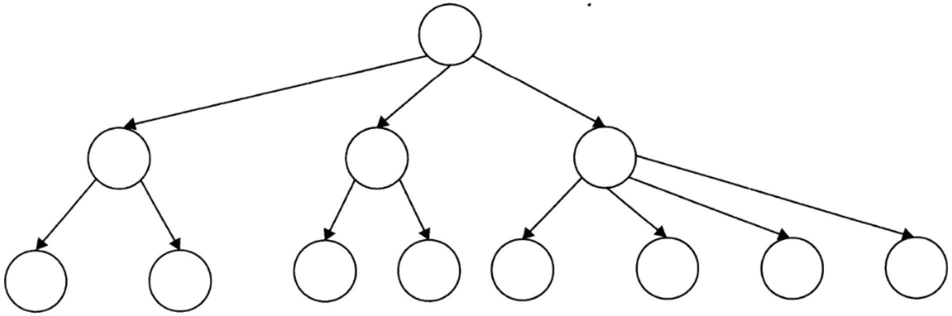
- Trees solve the prefix problem (e.g., find all terms starting with automat).
- Binary tree: Search is slightly slower than that in a hash, i.e., O(log M), where M is the size of the vocabulary. O(log M) only holds for balanced trees. Rebalancing binary trees is expensive.



3.1 Search structures for dictionaries

Trees

- B-trees
  - Definition: every internal node has a number of children in the interval [a, b], where a, b are two appropriate positive integers, e.g., [2,4]
  - Mitigate the rebalancing problem



Outline

- 3.1 Search structures for dictionaries
- 3.2 Wildcard queries
- 3.3 Spelling correction
- 3.4 Phonetic correction
- 3.5 References and further reading

3.2 Wildcard queries

Wildcard queries

- mon*: find all docs containing any term beginning with mon
  - Easy with B-tree dictionary: retrieve all terms t in the range: mon ≤ t < moo
- *mon: find all docs containing any term ending with mon
  - Maintain an additional tree for terms *backwards*
  - Then retrieve all terms t in the range: nom ≤ t < non

- Result: A set of terms that are matches for a wildcard query
- Then, we retrieve documents that contain any of these terms

3.2 Wildcard queries

How to handle * in the middle of a term

- Example: m*nchen
  - We could look up m* and *nchen in the B-tree and intersect the two sets of terms
  - Expensive
- Alternative: permuterm index
  - Basic idea: Rotate every wildcard query, so that the * occurs at the end
  - Store each of these rotations in the dictionary, i.e., in a B-tree

3.2 Wildcard queries

Permuterm index

- For term hello: add the following terms to the B-tree
  - *hello$:* a term ending with hello, and beginning with **NULL** (i.e., hello)
  - *ello$h:* any term ending with ello, and beginning with h
  - *llo$he:* any term ending with llo, and beginning with he
  - *lo$hel:* any term ending with lo, and beginning with hel
  - *o$hell:* any term ending with o, and beginning with hell
  - *$hello:* a term ending with **NULL**, and beginning with hello (i.e., hello)

- Note: $ is a special word **boundary** symbol

3.2 Wildcard queries

Permuterm -> term mapping

- *hello$* -> hello
- *ello$h* -> hello
- *llo$he* -> hello
- *lo$hel* -> hello
- *o$hell* -> hello
- *$hello* -> hello

3.2 Wildcard queries

Permuterm index

- Queries
  - For X (i.e., a term that is equal to X), look up X$ or $X
  - For X* (i.e., any term beginning with X), look up $X*
  - For *X (i.e., any term ending with X), look up X$*
  - For X*Y (i.e., any term ending with Y, and beginning with X), look up Y$X*
  - For *X* (i.e., any term containing X), look up X*
    - Notes: Any term ending with anything, and beginning with anything, and thus no $

3.2 Wildcard queries

Permuterm index

- Example: For hel*o, look up o$hel*
  - Any term ending with o, and beginning with hel
- Example: For *ell*, look up ell*
  - Any term containing ell

3.2 Wildcard queries

Processing a lookup in the permuterm index

- Step 1: Rotate a query wildcard to the right
- Step 2: Use B-tree lookup as before

- Problem: Permuterm more than quadruples the size of the dictionary compared to a regular B-tree (it is an empirical number)

- Permuterm index would better be called a permuterm tree, but permuterm index is the more common name

3.2 Wildcard queries

***k*-gram indexes**

- Enumerate all character *k*-grams (a sequence of *k* characters) occurring in a term
- 2-grams are called bigrams

- Example: from *April is the cruelest month*, we get the 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\$
  - \$ is a special word **boundary** symbol
  - Maintain an inverted index from bigrams to the terms that contain the corresponding bigrams

3.2 Wildcard queries

Postings list in a 3-gram inverted index

3.2 Wildcard queries

***k*-gram (bigram, trigram, . . .) indexes**

- We now have two different types of inverted indexes
  - The term-document inverted index for finding documents based on a query consisting of terms
  - The *k*-gram inverted index for finding terms based on a “query” consisting of *k*-grams

3.2 Wildcard queries

Processing wildcarded terms in a bigram index

- Query mon* can now be run as: $m AND mo AND on
- Gets us all terms with the prefix *mon* …
- ... but also many "false positives" like MOON.

- We must postfilter these terms against the “query”.
- Surviving terms are then looked up in the term-document inverted index.

- *k*-gram index vs. permuterm index
  - *k*-gram index is more space efficient.
  - Permuterm index doesn’t require postfiltering.

3.2 Wildcard queries

Exercise

- Why doesn't Google fully support wildcard queries?

3.2 Wildcard queries

Exercise

- Problem 1: We must potentially execute a large number of Boolean queries.
  - Very expensive, e.g., [gen* universit*]: geneva university OR general universities OR …

- Problem 2: Users hate to type.
  - This would significantly increase the cost of answering queries.

- Somewhat alleviated by Google Suggest.

Outline

- 3.1 Search structures for dictionaries
- 3.2 Wildcard queries
- 3.3 Spelling correction
- 3.4 Phonetic correction
- 3.5 References and further reading

3.3 Spelling correction

Edit distance (编辑距离)

- The edit distance between string s1 and string s2 is the minimum number of basic operations that convert s1 to s2.
- Levenshtein distance (莱文斯坦距离): The admissible basic operations are insert, delete, and replace
  - Levenshtein distance *dog-do*: 1
  - Levenshtein distance *cat-cart*: 1
  - Levenshtein distance *cat-cut*: 1
  - Levenshtein distance *cat-act*: 2
- Damerau-Levenshtein distance (达梅劳-莱文斯坦距离) includes transposition as a fourth possible operation.
  - Damerau-Levenshtein distance *cat-act*: 1

3.3 Spelling correction

Levenshtein distance: Computation

- Calculate the Levenshtein distance between strings "cats" and "fast".

		f	a	s	t
	0	1	2	3	4
c	1	1	2	3	4
a	2	2	1	2	3
t	3	3	2	2	2
s	4	4	3	2	3

3.3 Spelling correction

Each cell of Levenshtein matrix (1/2)

```
LevenshteinDistance(s1, s2)
for i ← 0 to |s1|
do m[i, 0] = i
for j ← 0 to |s2|
do m[0, j] = j
for i ← 1 to |s1|
do for j ← 1 to |s2|
    do if s1[i] = s2[j]
        then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
        else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
return m[|s1|, |s2|]
```

cost of getting here from my upper left neighbor (copy or replace)	cost of getting here from my upper neighbor (delete)
cost of getting here from my left neighbor (insert)	the minimum of the three possible "movements"; the cheapest way of getting here

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

3.3 Spelling correction

Each cell of Levenshtein matrix (2/2)

		f	a	s	t
	0	1	2	3	4
c	1	1 2 / 2 1	2 3 / 2 2	3 4 / 3 3	4 5 / 4 4
a	2	2 2 / 3 2	1 3 / 3 1	3 4 / 2 2	4 5 / 3 3
t	3	3 3 / 4 3	3 2 / 4 2	2 3 / 3 2	2 4 / 3 2
s	4	4 4 / 5 4	4 3 / 5 3	2 3 / 4 2	3 3 / 3 3

cost of getting here from my upper left neighbor (copy or replace)	cost of getting here from my upper neighbor (delete)
cost of getting here from my left neighbor (insert)	the minimum of the three possible "movements"; the cheapest way of getting here

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

3.3 Spelling correction

Dynamic programming

- The optimal solution to the problem contains within its subsolutions, i.e., optimal solutions to subproblems.
  - Subproblem in the case of edit distance: what is the edit distance of **two prefixes**
  - These subsolutions are computed over and over again when computing the global optimal solution in a **brute-force** algorithm.

3.3 Spelling correction

Using edit distance for spelling correction

- Step 1: Given a query, first enumerate all character sequences within a preset (possibly weighted) edit distance

- Step 2: Intersect this set with our list of “correct” words

- Step 3: Then suggest terms in the intersection to the user

3.3 Spelling correction

Weighted edit distance

- The weight of an operation depends on the characters involved
  - Meant to capture keyboard errors, e.g., *m* is more likely to be mistyped as *n* than as *q*. Therefore, replacing *m* by *n* is a smaller edit distance than by *q*.
- We now require a weight matrix as input.
- Modify dynamic programming to handle weights

3.3 Spelling correction

Exercise

- Compute Levenshtein distance matrix for “oslo” and “snow”

3.3 Spelling correction

- **Each cell of Levenshtein matrix (1/2)**

		s	n	o	w
	0	1	2	3	4
o	1	1 2 / 2 1	2 3 / 2 2	2 4 / 3 2	4 5 / 3 3
s	2	1 2 / 3 1	2 3 / 2 2	3 3 / 3 3	3 4 / 4 3
l	3	3 2 / 4 2	2 3 / 3 2	3 4 / 3 3	4 4 / 4 4
o	4	4 3 / 5 3	3 3 / 4 3	2 4 / 4 2	4 5 / 3 3

cost of getting here from my upper left neighbor (copy or replace)	cost of getting here from my upper neighbor (delete)
cost of getting here from my left neighbor (insert)	the minimum of the three possible "movements"; the cheapest way of getting here

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

3.3 Spelling correction

- **Each cell of Levenshtein matrix (2/2)**

		s	n	o	w
	0	1	2	3	4
o	**1**	1 2 / 2 1	2 3 / 2 2	2 4 / 3 2	4 5 / 3 3
s	2	1 2 / 3 **1**	2 3 / 2 2	3 3 / 3 3	3 4 / 4 3
l	3	3 2 / 4 2	2 3 / 3 **2**	3 4 / 3 3	4 4 / 4 4
o	4	4 3 / 5 3	3 3 / 4 3	2 4 / 4 **2**	4 5 / 3 **3**

cost	operation	input	output
1	delete	o	*
0	(copy)	s	s
1	replace	l	n
0	(copy)	o	o
1	insert	*	w

3.3 Spelling correction

Spelling correction

- Two principal uses
  - Correct documents being indexed
  - Correct user queries
- Isolated word spelling correction
  - Check each word on its own for misspelling
  - Will not catch typos resulting in correctly spelled words, e.g., *an asteroid that fell **form** the sky*
- Context-sensitive spelling correction
  - Look at surrounding words
  - Can correct **form/from** error above

3.3 Spelling correction

Correcting documents

- We are not interested in interactive spelling correction of documents (e.g., Microsoft Word) in this class.
- In IR, we use document correction primarily for OCR'ed documents (OCR: optical character recognition).
- The general philosophy in IR is: don’t change the documents.

3.3 Spelling correction

Correcting queries

- **Isolated word spelling correction**
  - Premise (前提) 1: There is a list of "correct words" from which the correct spellings come.
  - Premise 2: We have a way of computing the distance between a misspelled word and a correct word.

  - Simple spelling correction algorithm: return the "correct" word that has the smallest distance to the misspelled word.

3.3 Spelling correction

***k*-gram indexes for spelling correction**

- Enumerate all *k*-grams in the query term
- Example: bigram index, misspelled word *bordroom*
  - Bigrams: bo, or, rd, dr, ro, oo, om

- Use the *k*-gram index to retrieve **"correct"** words that match query term *k*-grams

- Threshold by the number of matching *k*-grams, e.g., only vocabulary terms that are very similar (e.g., retrieved many times, or differ by only a few *k*-grams , or Jaccard coefficient)

3.3 Spelling correction

k*-gram indexes for spelling correction: *bordroom

BO → aboard → about → boardroom → border

OR → border → lord → morbid → sordid

RD → aboard → ardent → boardroom → border

Question: Why is Isolated word spelling correction problematic?

3.3 Spelling correction

Context-sensitive spelling correction (1/2)

- How can we correct "form" in ***"flew form munich“***

- Hit-based spelling correction
  - Step 1: Retrieve "correct" terms close to each query term. "flea" for "flew", "from" for "form", "munch" for "munich"
  - Step 2: Try all possible resulting phrases as queries with one word "fixed" at a time
    - Try query "flea form munich"
    - Try query "flew from munich"
    - Try query "flew form munch"
    - The correct query "flew from munich" has the most **hits**

3.3 Spelling correction

Context-sensitive spelling correction (2/2)

- The "hit-based" algorithm we just outlined is not very efficient
- More efficient alternative: look at the "collection" of queries instead of the documents

3.3 Spelling correction

General issues in spelling correction (1/2)

- User interface
  - automatic vs. suggested correction
  - ***Did you mean*** only works for one suggestion
  - What about multiple possible corrections?

  - Tradeoff: simple vs. powerful UI

3.3 Spelling correction

General issues in spelling correction (2/2)

- Cost
  - Spelling correction is potentially expensive.
  - Avoid running on every query?
  - Maybe just on queries that match few documents.

  - Guess: Spelling correction of major search engines is efficient enough to be run on every query.

Outline

- 3.1 Search structures for dictionaries
- 3.2 Wildcard queries
- 3.3 Spelling correction
- 3.4 Phonetic correction
- 3.5 References and further reading

3.4 Phonetic correction

Soundex

- Soundex is the basis for finding phonetic (as opposed to orthographic 拼字正确) alternatives.
- Example: *chebyshev / tchebyscheff*
- 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

3.4 Phonetic correction

Soundex algorithm (1/2)

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

3.4 Phonetic correction

Soundex algorithm (2/2)

- **Step 4**: Repeatedly remove one out of each pair of consecutive identical digits
- **Step 5**: Remove all zeros from the resulting string; pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits

3.4 Phonetic correction

Example: Soundex of *HERMAN*

- Step 1: Retain H
- Step 2: *ERMAN -> 0RM0N*
- Step 3: *0RM0N -> 06505*
- Step 4: *06505 -> 06505*
- Step 5: *06505 -> 655*
- Return *H655*
- Note: *HERMANN* will generate the same code

3.4 Phonetic correction

How useful is Soundex?

- Not very useful for information retrieval
- Ok for "high recall" tasks in other applications (e.g., International Criminal Police Organization)
- Zobel and Dart (1996) suggest better alternatives for phonetic matching in IR

Summary

- 3.1 Search structures for dictionaries
- 3.2 Wildcard queries
- 3.3 Spelling correction
- 3.4 Phonetic correction
- 3.5 References and further reading