Information Retrieval

Dictionaries and and tolerant retrieval

Hamid Beigy

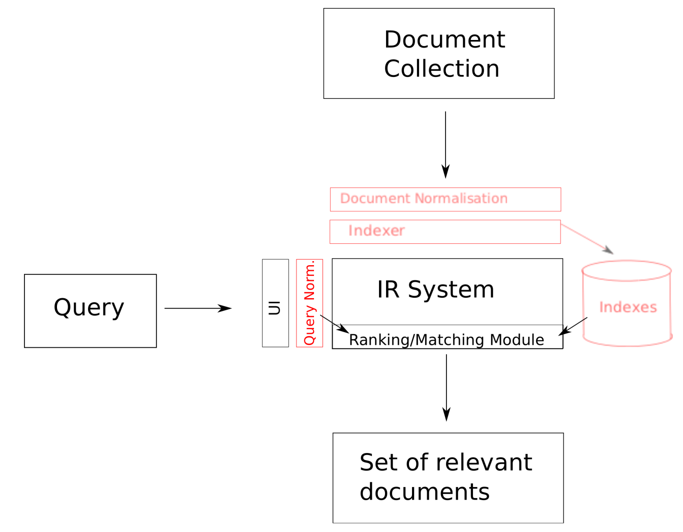
Sharif university of technology

October 9, 2019

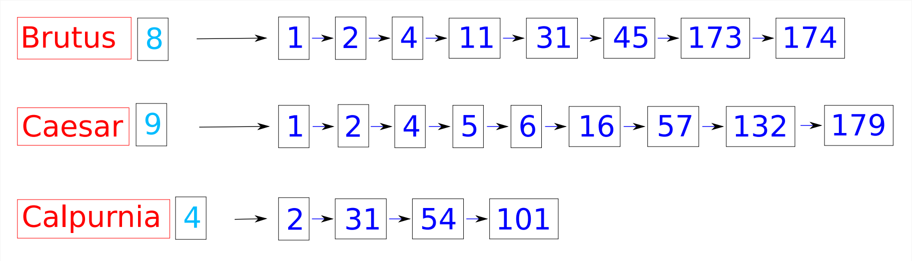
Table of contents

Table of contents

Information retrieval system components



Inverted index



This session

1. Data structures for dictionaries
   - Hash tables
   - Trees
   - *k*-term index
   - Permuterm index
2. Tolerant retrieval: What to do if there is no exact match between query term and document term
3. Spelling correction

Term-document incidence matrix

1. Inverted index

For each term t, we store a list of all documents that contain t.

BRUTUS $\longrightarrow$ 1 | 2 | 4 | 11 | 31 | 45 | 173 | 174

CAESAR $\longrightarrow$ 1 | 2 | 4 | 5 | 6 | 16 | 57 | 132 | ...

CALPURNIA $\longrightarrow$ 2 | 31 | 54 | 101

$\vdots$

dictionary **postings**

Dictionaries

1. Dictionary: the data structure for storing the term vocabulary.
2. Term vocabulary: the data
3. For each term, we need to store a couple of items:
   - document frequency
   - pointer to postings list
4. How do we look up a query term q in the dictionary at query time?

Data structures for looking up terms

1. Two different types of implementations:
    - hash tables
    - search trees
2. Some IR systems use hash tables, some use search trees.
3. Criteria for when to use hash tables vs. search trees:
    - How many terms are we likely to have?
    - Is the number likely to remain fixed, or will it keep growing?
    - What are the relative frequencies with which various terms will be accessed?

Table of contents

Hash tables

1. Hash table: an array with a hash function
   - Input: a key which is a query term
   - output: an integer which is an index in array.
   - Hash function: determine where to store / search key.
   - Hash function that minimizes chance of collisions.
     Use all info provided by key (among others).
2. Each vocabulary term (key) is hashed into an integer.
3. At query time: hash each query term, locate entry in array.

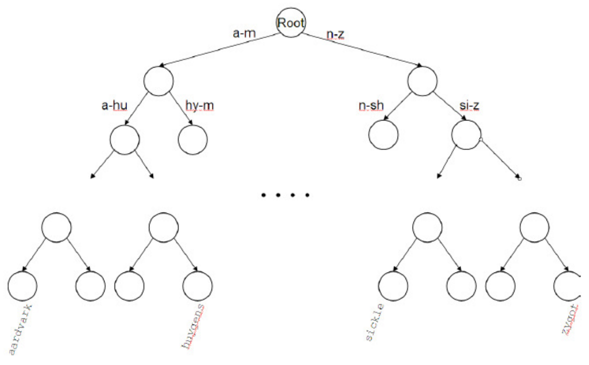
Hash tables

1. Advantages
   - Lookup in a hash is faster than lookup in a tree. (Lookup time is constant.)
2. disadvantages
   - No easy way to find minor variants (*résumé* vs. resume)
   - No prefix search (all terms starting with automat)
   - Need to rehash everything periodically if vocabulary keeps growing
   - Hash function designed for current needs may not suffice in a few years' time

Table of contents

Binary search tree

1. Simplest search tree: binary search tree



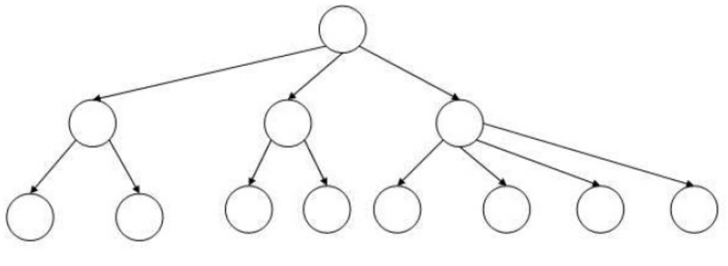
2. Partitions vocabulary terms into two subtrees, those whose first letter is between a and m, and the rest (actual terms stored in the leafs).
3. Anything that is on the left subtree is smaller than what's on the right.

Binary search tree

1. Cost of operations depends on height of tree.
2. Keep height minimum / keep binary tree balanced: for each node, heights of subtrees differ by no more than 1.
3. $O(\log M)$ search for balanced trees, where M is the size of the vocabulary.
4. Search is slightly slower than in hashes
5. But: re-balancing binary trees is expensive (insertion and deletion of terms).

B-Tree

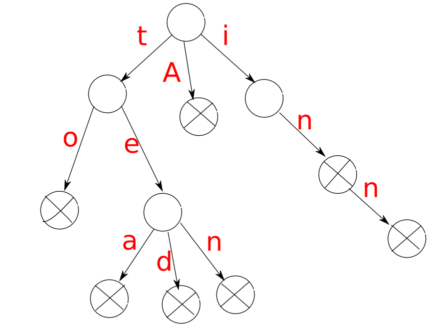
1. Need to mitigate re-balancing problem allow the number of sub-trees under an internal node to vary in a fixed interval.
2. B-tree definition: every internal node has a number of children in the interval [a, b] where a, b are appropriate positive integers, e.g., [2, 4].



3. Every internal node has between 2 and 4 children.

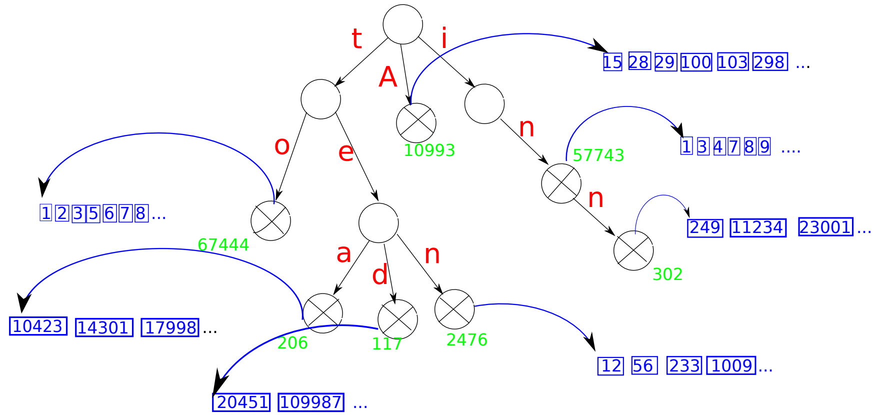
Trie

1. Trie is a search tree



2. An ordered tree data structure for strings
   - A tree where the keys are strings (keys tea, ted)
   - Each node is associated with a string inferred from the position of the node in the tree (node stores bit indicating whether string is in collection)

Trie in IR


t
i
A
15 28 29 100 103 298 ...
o
e
10993
n
57743
1 3 4 7 8 9 ....
1 2 3 5 6 7 8 ...
n
249 11234 23001 ...
67444
a
d
n
302
10423 14301 17998 ...
206
117
2476
12 56 233 1009 ...
20451 109987 ...

Wildcard queries

1. Query :hel*
2. Find all docs containing any term beginning with hel
3. Easy with trie: follow letters h-e-l and then lookup every term you find there
4. Query : *hel
5. Find all docs containing any term ending with hel
6. Maintain an additional trie for terms backwards
7. Then retrieve all terms in subtree rooted at l-e-h
8. In both cases:
   - This procedure gives us a set of terms that are matches for the wildcard queries
   - Then retrieve documents that contain any of these terms

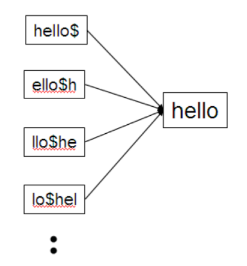
How to handle * in the middle of a term

1. Query: hel*o
2. We could look up hel* and *o in the tries as before and intersect the two term sets (expensive!).
3. Solution: permuterm index special index for general wildcard queries

Table of contents

Permuterm index

1. For term hello$ (given $ to match the end of a term), store each of these rotations in the dictionary (trie):
   hello$, ello$h, llohe, lohel, o$hell, $hello : permuterm vocabulary
2. Rotate every wildcard query, so that the * occurs at the end: for hel*o$, look up o$hel*



3. Problem: Permuterm more than quadrupels the size of the dictionary compared to normal trie (empirical number).

Table of contents

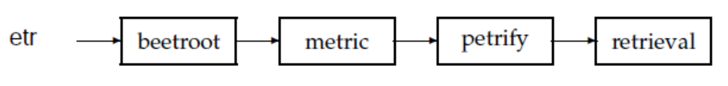
k-gram indexes

1. More space-efficient than permuterm index
2. Enumerate all character k-grams (sequence of k characters) occurring in a term and store in a dictionary

Example (Character bi-grams from April is the cruelest month)

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

3. $ special word boundary symbol
4. A postings list that points to all vocabulary terms containing a k-gram



k-gram index

1. Note that we have two different kinds of inverted indexes:
   - The term-document inverted index for finding documents based on a query consisting of terms
   - The k-gram index for finding terms based on a query consisting of k-grams

Processing wildcard queries in a (char) bigram index

1. Query hel* can now be run as:
   \$H AND HE AND EL
2. This will show up many false positives like blueheel.
3. Post-filter, then look up surviving terms in termdocument inverted index.
4. k-gram vs. permuterm index
   - k-gram index is more space-efficient
   - permuterm index does not require post-filtering.

Table of contents

Spelling correction

1. Query: an asterorid that fell form the sky
2. Query: britney spears
   queries: britian spears, britneys spears, brandy spears, prittany spears
3. In an IR system, spelling correction is only ever run on queries.
4. Two different methods for spelling correction:
   - Isolated word spelling correction
     Check each word on its own for misspelling
     Will only attempt to catch first typo above
   - Context-sensitive spelling correction
     Look at surrounding words
     Should correct both typos above

Isolated word spelling correction

1. There is a list of correct words for instance a standard dictionary (Websters, OED. . .)
2. Then we need a way of computing the distance between a misspelled word and a correct word
   - for instance Edit/Levenshtein distance
   - k-gram overlap
3. Return the correct word that has the smallest distance to the misspelled word.
   informaton ⇒ information

Edit distance

1. Edit distance between two strings s_1 and s_2 is defined as the minimum number of basic operations that transform s_1 into s_2.
2. Levenshtein distance: Admissible operations are insert, delete and replace
3. Example

dog	do	1 (delete)
cat	cart	1 (insert)
cat	cut	1 (replace)
cat	act	2 (delete+insert)

Distance matrix

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

Example: Edit Distance oslo snow

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

Each cell of Levenshtein matrix

Cost of getting here from my upper left neighbour (by copy or replace)	Cost of getting here from my upper neighbour (by delete)
Cost of getting here from my left neighbour (by insert)	Minimum cost out of these

Levenshtein matrix : An example

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

Example: (2, 2):

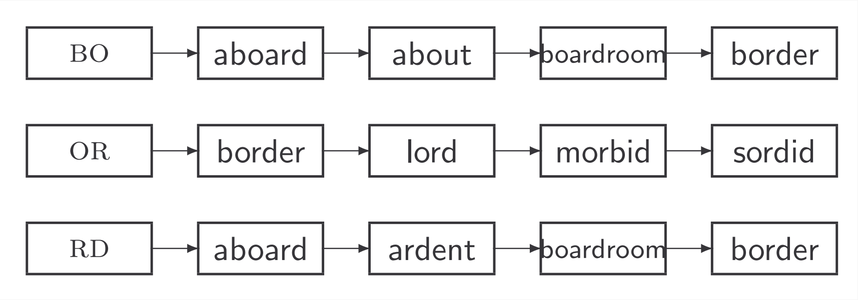
- Upper left: cost to replace “o” to “s” (cost: 0+1)
- Upper right: come from above where I have already inserted “s”: all I need to do is delete “o” (cost: 1+1)
- Bottom left: come from left neighbour where I have deleted “o”: all I need to do is insert “s” (cost: 1+1)

Using edit distance for spelling correction

1. Given a query, enumerate all character sequences within a pre-set edit distance.
2. Intersect this list with our list of correct words.
3. Suggest terms in the intersection to user.
4. Cons
   - Comparing query term q to all terms in the vocabulary is too expensive
   - Solution: use heuristics to determine subset

k-gram indexes for spelling correction

1. Enumerate all k-grams in the query term
2. Misspelled word: bordroom
3. Use k-gram index to retrieve correct words that match query term k-grams
4. Threshold by number of matching k-grams
5. Eg. only vocabularly terms that differ by at most 3 k-grams



Context-sensitive spelling correction

1. An idea: hit-based spelling correction
   flew form munich
2. Enumerate corrections of each of the query terms
   flew ⇒ flea
   form ⇒ from
   munich ⇒ munch
3. Holding all other terms fixed, try all possible phrase queries for each replacement candidate
   flea form munich ⇒ 62 results
   flew from munich ⇒ 78900 results
   flew form munch ⇒ 66 results
4. Not efficient. Better source of information: large corpus of queries, not documents

Table of contents

Soundex

- Soundex is the basis for finding phonetic (as opposed to orthographic) 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

Soundex algorithm

1. Retain the first letter of the term.
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 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
4. Repeatedly remove one out of each pair of consecutive identical digits
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

Example: Soundex of *HERMAN*

- Retain H
- *ERMAN* → *0RM0N*
- *0RM0N* → *06505*
- *06505* → *06505*
- *06505* → *655*
- Return *H655*
- Note: *HERMANN* will generate the same code

How useful is Soundex?

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

Reading

Please read chapter 3 of Information Retrieval Book.