Information Retrieval

Dictionaries and and tolerant retrieval

Hamid Beigy

Sharif university of technology

October 6, 2018
Table of contents

1. Introduction
2. Hash tables
3. Search trees
4. Permuterm index
5. k-gram indexes
6. Spelling correction
Table of contents

1 Introduction

2 Hash tables

3 Search trees

4 Permuterm index

5 k-gram indexes

6 Spelling correction
Information retrieval system components

- Query
- Document Collection
  - Document Normalisation
  - Indexer
- IR System
  - Ranking/Matching Module
- Set of relevant documents

Today: more indexing, some query normalisation
Inverted index

Brutus 8

Caesar 9

Calpurnia 4
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
## Inverted Index

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Documents Containing $t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1, 2, 4, 11, 31, 45, 173, 174</td>
</tr>
<tr>
<td>Caesar</td>
<td>1, 2, 4, 5, 6, 16, 57, 132, ...</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2, 31, 54, 101</td>
</tr>
</tbody>
</table>

...
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 qi in the dictionary at query time?
Two different types of implementations:

- hash tables
- search trees

Some IR systems use hash tables, some use search trees.

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

1. Introduction
2. Hash tables
3. Search trees
4. Permuterm index
5. k-gram indexes
6. Spelling correction
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

1 Introduction

2 Hash tables

3 Search trees

4 Permuterm index

5 k-gram indexes

6 Spelling correction
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
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

1. Introduction
2. Hash tables
3. Search trees
4. Permuterm index
5. k-gram indexes
6. Spelling correction
Permuterm index

1. For term \(\text{hello}$\) (given $ to match the end of a term), store each of these rotations in the dictionary (trie):
   \(\text{hello}$, \(\text{ello}$h, \(\text{llo}$he, \(\text{lo}$hel, \(o$hell, \$hello\) : permuterm vocabulary

2. Rotate every wildcard query, so that the * occurs at the end: for \(\text{hel}$o$\), look up \(o$hel*$

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

1 Introduction
2 Hash tables
3 Search trees
4 Permuterm index
5 k-gram indexes
6 Spelling correction
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)

$a$ $a$ $p$ $p$ $r$ $r$ $i$ $i$ $l$ $l$ $i$ $i$ $s$ $s$ $t$ $t$ $h$ $h$ $e$ $e$ $e$ $c$ $c$ $r$ $r$ $u$ $u$ $e$ $e$ $l$ $l$ $e$ $e$ $s$ $s$ $t$ $t$ $m$ $m$ $m$ $m$ $o$ $o$ $n$ $n$ $n$ $n$ $t$ $t$ $h$ $h$

3. $\$$ special word boundary symbol
4. A postings list that points to all vocabulary terms containing a k-gram
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
Query `hel*` can now be run as:

```
$h \text{ AND } he \text{ AND } el$
```

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

1. Introduction
2. Hash tables
3. Search trees
4. Permuterm index
5. k-gram indexes
6. Spelling correction
Spelling correction

1. Query: an *asteroid* 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 $\Rightarrow$ 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

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>n</th>
<th>o</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>o</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>l</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>o</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Example: Edit Distance

```
oslo
  0 1 1 2 2 3 3 4 4
  1 2 3 2 4 4 5
  1 2 3 2 3 3 3
  2 3 3 3 5
  3 4 4 4 4
  4 5
```

<table>
<thead>
<tr>
<th>cost</th>
<th>operation</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>delete</td>
<td>o</td>
<td>*</td>
</tr>
<tr>
<td>0</td>
<td>(copy)</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>1</td>
<td>replace</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>0</td>
<td>(copy)</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>1</td>
<td>insert</td>
<td>*</td>
<td>w</td>
</tr>
<tr>
<td>Cost of getting here from my upper left neighbour (by <strong>copy</strong> or <strong>replace</strong>)</td>
<td>Cost of getting here from my upper neighbour (by <strong>delete</strong>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cost of getting here from my left neighbour (by <strong>insert</strong>)</td>
<td>Minimum cost out of these</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Levenshtein matrix: An example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>s</th>
<th>n</th>
<th>o</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>o</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>s</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>o</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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 vocabulary terms that differ by at most 3 k-grams

---

**Diagram:**

- BO: aboard → about → boardroom → border
- OR: border → lord → morbid → sordid
- RD: aboard → ardent → boardroom → border
An idea: hit-based spelling correction
flew form munich

Enumerate corrections of each of the query terms
flew ⇒ flea
form ⇒ from
munich ⇒ munch

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

Not efficient. Better source of information: large corpus of queries, not documents
Please read chapter 3 of Information Retrieval Book.