Chapter 3:
Dictionaries and Tolerant Retrieval

Reading:

- 3.1
- 3.2
- 3.3

Now we have a good **index** for recording occurrence of terms in a corpus. How else can we improve our search engines?

Tolerance: automatically adjust to small errors which we can determine a plausible correction for.

Data structures for dictionaries

First goal when satisfying a query: find the postings for the terms in the query. Associating a postings list with a term is a job for a **dictionary/map** data structure.

Questions for choosing a structure:

1. How many keys are we likely to have?
2. Is that number likely to stay static or change a lot?
3. What frequencies of access for each key?

Hash maps:

Use a hash table and hasing algorithm to enter a term into an array, and as the value of that array store a pointer to a list of postings.

Pro: very fast $\Theta(1)$ amortized insertion and retrieval. Can handle large numbers of keys with special structures.

Con: keys not stored in any predictable or relevant order. Frequent keys cannot be retrieved faster than infrequent keys. Hash functions can have more collisions as vocab size increases.

Search trees:

Use a tree to store the vocabulary; in a term's tree node, store a pointer to the term's list.

Pro: structure reflects relations between terms; easy to find prefix words

Con: slower $O(\log n)$ insertion and retrieval if tree is balanced. Rebalancing takes time. Terms must be **absolutely orderable**.

Wildcard queries

Wildcard query: use a `*` character to indicate "any sequence of characters".

- Unsure of spelling: `sidney` or `sydney`? `s*dney`!
- Aware of multiple spellings: `color` or `colour`? `colo*r`!
- Not sure if stemming is used: `swimming` or `swimmer`? `swim*`!

Trailing wildcards:

Given a search term of `q*`, where `q` is a sequence of characters:

1. Traverse a search tree of the vocabulary until finding the prefix `q`

2. All W terms in the right subtree have prefix `q`.
3. Run an OR query with all W terms.

Leading wildcards:

Given a search term of `*q`:

1. Build a **reverse search tree** using the reverse of teach term in the vocabulary.
2. Traverse this tree looking for the reverse of `q`.
3. Run an OR query.

Single wildcards:

Given a search term `q1*q2`:

1. Do a lookup of `q1` in the search tree, and a lookup of reverse of `q2`.
2. Intersect.
3. OR.

General wildcard queries:

Permuterm index:

1. Put a `$` at the end of each term in the vocabulary.
2. Build an index that links each rotation of the tem back to the original term.

Example 1. Build permuterm rotations for `mourn`

3. Consider the wildcard query `m*n`. Rotate the term so that `*` is at the end. `m*n$ -> n$m*`. Find this term in the permuterm index, loading all real terms with this rotation. Finds `men`, `moron`, `mourn`.
4. For multiple stars, rotate right until one star is at end, then do a lookup on everything up to the first star. `fi*mo*er$ -> er$fi*mo* -> er$fi*`. Find this term in the index, which will contain some terms that do not match original query. Filter based on original query.

Weaknesses: massive increase in vocabulary size.

k-gram index: A **k-gram** is a sequence of k characters from a term. `ast` and `tle` are 3-grams of `castle`. Adding `$` to the beginning and end of the term, we get these 3-grams: `$ca cas ast stl tle le$`.

In a k-gram index:

1. The dictionary contains all k-grams from all terms in the vocabulary.
2. Each index entry maps to all vocabulary terms with that k-gram. Example: `etr -> metric, retrieval`
3. Consider query `re*ve`: run 3-gram query `$re AND ve$ -> relive, revive, remove, retrieve`
4. **Problem**: consider `red*`; run 3-gram query `$re AND red -> retired`, which does not fit original query.
5. Run a **post-filtering step**: from left to right in each candidate term, match each of the substrings from the query.

Spelling Correction

Want: spelling correction for **queries**. Suppose we can magically determine whether a word is misspelled. Then we can:

1. Of various alternate "correct" spellings for `q`, choose the "nearest" one. Requires a notion of "nearness" for words.

2. When multiple spellings are equidistant, select the one that is more common. (How?) Example: `grunt` and `grant` equidistant for `grnt`; which one to select?
    (a) Frequency: choose word that is more frequent in collection
    (b) User feedback: choose word that is more commonly searched

There are options for exposing spelling correction to the user. Suppose the user searches `carot`:

1. Retrieve all documents with `carot` as well as **any** spell-corrected term of `carot`, like `carrot` or `tarot`
2. As in (1), but only if `carot` is not in the vocabulary
3. As in (1), but only if the query returned below some threshold number of documents
4. When returning less than some threshold, display the results, and offer a spelling suggestion to the user: "Did you mean *carrot*?"

Isolated-term correction:

Edit distance: a measure of the number of "edit operations" to turn string s_1 into string s_2.

- insert a character
- remove a character
- replace a character with another

In this scheme, called Levenshtein Distance.

Levenshtein Distance algorithm: Let u and v be strings, with $|u| = m$ and $|v| = n$. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define $d(i, j)$ as the edit distance between the first i letters of u and the first j letters of v, so that $d(m, n)$ is the edit distance between u and v themselves. Then

$$d(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ \min\left(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+(u_i \neq v_j)\right) & \end{cases}$$

Example 2. Show the recursive evaluation of $d(m, n)$ for $u =$`fries`, $v =$`frys`. Then find the Levenshtein distance for `cats` and `fast`.

Then find the Levenshtein distance for `squirrel` and `skwirel`

Finding suitable corrections: Suppose the user enters `gost` as a query. There are many terms that could be corrections: `ghost`, `goat`, `host`, `most`, ... We can't test every word in the vocabulary to find all possible replacements... need a better system to identify potential corrections.

- **assume they got the first letter right**: fairly safe assumption, but not perfect.
- **k-gram overlap**: find the k-grams of the query term. Select all vocabulary terms that have some % of kgrams in common. Find edit distance of those terms, use the lowest distance.
  Problem: "`hostess`" – is that a reasonable correction?
- **Jaccard coefficient**:
  - collect all terms with at least one k-gram of `q`'s.
  - for each term, calculate the Jaccard coefficient $= \frac{|A \cap B|}{|A \cup B|}$.
  - how to get those kgrams? already have `q`'s (A); for each term t, we don't actually need the kgrams! Already know $|A \cap B|$; use $|A \cup B| = |A| + |B| - |A \cap B|$