Search Engines

Gertjan van Noord

September 5, 2024

Last week

- Introduction on Information Retrieval / Search Engines
- Boolean Retrieval
- Posting Lists
- Intersection of ordered lists
- Efficiency: linear versus quadratic

Last week

- Efficiency: linear versus quadratic
- Merging algorithm in Python
- Sets in Python
- Not all programming languages have (efficient) sets
- Sometimes you do not need intersection, but something very similar
- Efficient sets and dictionaries are built by means of *hashes*
  - Hashes require more storage than ordered lists

Disadvantage of hashes

- used by Python to implement dictionaries and sets
- to store N elements, it reserves space for x*N elements, where x is between 30 and 60
- for lists, x is between 8 and 9

Length and memory size

- function `len()` returns the number of elements of a container (list, dict, set, . . .)
- function `sys.getsizeof()` on the other hand returns the memory size of the object

Disadvantage of hashes

```
import sys

d = {}
for i in range(1,100000000):
     d[str(i)]=i
     if not len(d) % 10000:
          print(f'{len(d)}, sys.getsizeof(d) = \
               {sys.getsizeof(d)} {sys.getsizeof(d)/len(d)'})
```

Disadvantage of hashes

```
10000, sys.getsizeof(d) = 295000 29.5
20000, sys.getsizeof(d) = 589920 29.496
30000, sys.getsizeof(d) = 1310808 43.6936
40000, sys.getsizeof(d) = 1310808 32.7702
50000, sys.getsizeof(d) = 2621536 52.43072
60000, sys.getsizeof(d) = 2621536 43.69226666666667
70000, sys.getsizeof(d) = 2621536 37.45051428571428
80000, sys.getsizeof(d) = 2621536 32.7692
90000, sys.getsizeof(d) = 5242968 58.2552
100000, sys.getsizeof(d) = 5242968 52.42968
110000, sys.getsizeof(d) = 5242968 47.66334545454546
120000, sys.getsizeof(d) = 5242968 43.6914
130000, sys.getsizeof(d) = 5242968 40.33052307692308
140000, sys.getsizeof(d) = 5242968 37.449771428571432
150000, sys.getsizeof(d) = 5242968 34.95312
160000, sys.getsizeof(d) = 5242968 32.76855
170000, sys.getsizeof(d) = 5242968 30.84098823529411 6
180000, sys.getsizeof(d) = 10485856 58.254755555555555
```

```
190000, sys.getsizeof(d) = 10485856 55.18871578947368
200000, sys.getsizeof(d) = 10485856 52.42928
210000, sys.getsizeof(d) = 10485856 49.93264761904762
220000, sys.getsizeof(d) = 10485856 47.66298181818182
...
```

Disadvantage of hashes

```
import sys

d = []
for i in range(1,100000000):
     d.append(i)
     if not len(d) % 10000:
          print(f'{len(d)}, sys.getsizeof(d) = \
               {sys.getsizeof(d)} {sys.getsizeof(d)/len(d)}')
```

Disadvantage of hashes

```
10000, sys.getsizeof(d) = 87616 8.7616
20000, sys.getsizeof(d) = 178016 8.9008
30000, sys.getsizeof(d) = 253624 8.454133333333333
40000, sys.getsizeof(d) = 321096 8.0274
50000, sys.getsizeof(d) = 406488 8.12976
60000, sys.getsizeof(d) = 514560 8.576
70000, sys.getsizeof(d) = 578928 8.2704
80000, sys.getsizeof(d) = 651344 8.1418
90000, sys.getsizeof(d) = 732808 8.142311111111111
100000, sys.getsizeof(d) = 824456 8.24456
110000, sys.getsizeof(d) = 927560 8.432363636363636
120000, sys.getsizeof(d) = 1043552 8.696266666666666
130000, sys.getsizeof(d) = 1043552 8.027323076923077
140000, sys.getsizeof(d) = 1174040 8.386
150000, sys.getsizeof(d) = 1320840 8.8056
160000, sys.getsizeof(d) = 1320840 8.25525
...
```

This week: chapter 2

- Term vocabulary, Tokenization, Normalization
- Phrase Queries
- Posting lists with positions
- Excursion: suffix array

Preprocessing of documents

- choose the unit of indexing: typically words
- tokenization: from text to a sequence of words
- stop list?
- normalization?

Normalization

- Case-folding
- Diacritics
- Stemming / Lemmatization
- Decompounding
- Variant spellings
- Multi-word units
- . . .

Tokens, types, terms

- token: each occurrence of a word in the text
- type: each distinct word in the text
- term: is included in the index, normalized version of the token/type

Tokens, types, terms

Suppose the only normalization we do is: everything in lower case.

"The Lord of the Rings missed the Ring"

- tokens: The, Lord, of, the, Rings, missed, the, Ring
- terms: the, lord, of, the, rings, missed, the, ring

Tokens, types, terms

Suppose the only normalization we do is: everything in lower case, and use lemma for given word

"The Lord of the Rings missed the Ring"

- tokens: The, Lord, of, the, Rings, missed, the, Ring
- terms: the, lord, of, the, ring, miss, the, ring

Normalization

- $terms$ are equivalence classes over $tokens$
- In preprocessing, each $token$ is mapped to its $term$
- In the query, each $token$ is mapped to its $term$

Thus: if you search for a token, you get hits for all variants of that token

Case-folding

- Case folding: case distinction is removed
- "de" versus "DE" (Douwe Egberts)
- Truecasing: remove case only if irrelevant, e.g., if first word in sentence; names remain capitalized
- As usual, the more intelligent solution is more difficult than you might think because there always is ambiguity
- *Ben ik te min / Ben is te min*
- Piet komt wel. En Wil?. *Wil niet*.
- Lukt het? Nee. *Wil niet*.

Diacritics, Spelling variants

- zeeen → zeeën
- zeeën → zeeën
- onmiddellijk → onmiddellijk
- onmidellijk → onmiddellijk
- Both for indexing and query?

Inflection, Derivation

- Inflection: Changing a word to express person, case, tense, aspect
- sleep $\rightarrow$ sleeps
- box $\rightarrow$ boxes
- Derivation: Formation of a new word
- to browse $\rightarrow$ browser
- slow $\rightarrow$ slowly

Stemming and lemmatizing

- *verb forms*: inform, informs, informed, informing
- *derivation*: information, informative, informal (?)
- *stem*: inform
- *lemma*: inform, information, informative, informal

Stemming and lemmatizing

- *verb forms*: sing, sings, sang, sung, singing
- *derivation*: singer, singers, song, songs
- *stem*: sing, sang, sung, song
- *lemma*: sing, singer, song

Why is stemming often used whereas lemmatizing is more precise?

- Lemmatizing needs access to vocabulary
- Lemmatizing needs morphological analysis
- . . . and sometimes syntactic analys

Decompounding

If the user searches for “marketing”, is he interested in hits such as “marketingjargon”?

- Decompounding: compound is split in two (or more) words
- Not so relevant for English where compounds are not written as one word
- Potentially important for German, Dutch, . . .
- What is the effect of decompounding on *Recall* and *Precision* for the query “marketing”
- What to do with the query “marketingjargon”?

Decompounding

Not always easy . . .

stiefouderadoptie	stief+oude+rad+optie
bestuursapparaat	bestuur+sap+paraat
overbeharing	over+beha+ring
mensenrechtengroepen	mensen+recht+eng+roe+pen
partijkader	part+ijk+ader
reinigingsmiddelen	reiniging+smid+delen
rotspartij	rot+spar+tij
theaterspektakel	theater+pek+takel
zonnestroom	zon+nest+room
haptonoom	hap+ton+oom
volksoproer	volk+sop+roer
plantenteelt	plan+tent+eelt

Phrase Queries

- search for *"information retrieval"* is not the same as *information AND retrieval*

- various approaches:
  - biword index and phrase index
  - positional index
  - suffix array (later!)
  - . . .

Biword index

- Every sequence of two words is a term
- Any phrase of two words can be queried
- Longer phrases can be broken down
- "Informatiekunde Rijksuniversiteit Groningen" $\longrightarrow$
  "Informatiekunde Rijksuniversiteit" AND "Rijksuniversiteit Groningen"
- Heuristics to limit the number of terms
- Same for longer terms: phrase index

Biword index and Phrase index

- Very many biwords: index too large
- Limit biwords: less effective for some queries

Positional index

Add in the posting lists for each document: the positions of the term

```
< information , < < Doc1,< 1,4,22,35 > >,
                  < Doc4,< 5,17,30 > >
                >
  retrieval ,   < < Doc1,< 5,20  > >,
                  < Doc3,< 2 > >,
                  < Doc4,< 18,31 > >
                >
>
```

Using data structures that allow for efficient

- finding documents that contain both terms
- establishing that for a document the terms occur as phrase

How to use the positional index

- For a query such as “information retrieval”, take the posting list of “information” and the posting list of “retrieval”
- Find matching documents
- For a matching document, find ‘matching’ positions (how?)
- Very similar to intersection, but . . .
- Also works for longer phrases

Phrase queries: combination schemes

- Use phrase index for frequently used phrase queries (“New York”, “Michael Jackson”)
- in particular if the individual words are very frequent (“The the”)
- Use positional index for other phrase queries
- For very long phrase queries, use a suffix array

Suffix array

- Very simple, but interesting data structure useful for several string problems.
- Used in:
  - text search
  - plagiarism detection
  - data compression
  - computational biology
  - large language models

Suffix array

- Searching very long phrase queries
- What is the longest repetition in a given text
- What are very long identical passages in two given texts
- What is most frequent 12-gram (sequence of 12 words) in a given text

Suffix array

Wikipedia: A suffix array is a sorted array of all suffixes of a string.

Suffix array

String: informatie

Suffixes:

```
informatie
nformatie
formatie
ormatie
rmatie
matie
atie
tie
ie
e
```

Ordered:

```
atie
e
formatie
ie
informatie
matie
nformatie
ormatie
rmatie
tie
```

Suffix Array

Represent each suffix $c_i \ldots c_n$ by its starting position i Suffix:

0
1
2
3
4
5
6
7
8
9

Ordered:

6
9
2
8
0
5
1
3
4
7

Suffix Array

Suppose we create a suffix array for `small.txt` (ignoring the keys), and then print the first characters of each suffix in the order of the suffix array:

```
aajee aajoo " .\nIn het dagelijks leven is hij vri
aajoo " .\nIn het dagelijks leven is hij vrijgeves
aak ( " Acer campestre " ) is een plant uit de ze
aak .\nDe Spaanse aak ( " Acer campestre " ) is ee
aak herfstkleuren .\nDe soort kan eenhuizig of twe
aak wordt tot 10 m hoog .\nDe plant wordt vaak als
aal ' dan wel met ' paling ' worden aangeduid .\nE
aal ( " Anguilla anguilla " ) , is een straalvinn
aal en de roofzuchtige murene , alle soorten zijn
aalmoes of smeergeld functioneert .\nDe term is al
aalmoezenier , maar bleef doorstuderen .\nVlad Dra
aalmoezenier en koormeester van de Londense St. P
aalscholver ( " Phalacrocorax auritus " ) ( Doubl
aalscholver ( " Phalacrocorax carbo " ) , ook wel
aalscholver .\nDe Amerikaanse aalscholver ( " Phal
aalscholver behoort tot de familie van de aalscho
```

aalscholver uit de familie van fuutkoeten .\nUiter
aalscholvers (Phalacrocoracidae) , waarvan 36 s
aalscholvers .\nHet was het eerste gebied dat de v
aalscholvers en een aantal sterns .\nEtheenoxide .
aaltjes " behoren tot de nematoden .\nNematologie
aambeeld (of aanbeeld) is een gereedschap .\nHet
aambeeld .\nEen aambeeld (of aanbeeld) is een ge
aambeeld .\nSoms is de gehele troposfeer onstabiel
aambeeld er tien dagen over zou doen om van het o
aambeeld zou tien dagen nodig hebben om vanaf Our
aan\nFrans .\nHet Frans (" français ") behoort t
aan\nIJshockey valt onder de " balsporten " maar d
aan\nInterferentie van elektronen door Claus Jöns
aan\nMineraal .\nEen mineraal is een stof die in ho
aan\nNog anders geformuleerd , een basis van vecto
aan ") en het verbreken van een verbinding een l

Suffix Array

- Searching very long phrase queries:

  do a binary search in the suffix array to find the first suffix which starts with the query

Suffix Array

- What is the longest repetition in a given text:

  one pass through the suffix array and keep track of the longest identical prefix seen so far

Suffix Array

- What are very long identical passages in two given texts:

  do a kind of merge sort of the two suffix arrays of both texts and report suffixes with long identical prefixes

Suffix Array

- What is most frequent 12-gram (sequence of 12 words) in a given text:

  one pass through the suffix array and keep track of the most frequent 12-gram prefix seen so far

Suffix Array

- A suffix array for a given text can be constructed
  - in linear time
  - without much additional memory
- the naive method is slow, but is a one-liner in Python

```
def construct_sufarr(corpus):
    return sorted(range(len(corpus)), key=lambda el: corpus[el:])
```

- there are Python libraries with efficient implementation

Exercises

Part A: Preparation for Phrase queries

The assignment builds further on the assignment of week 1, using the same Wikipedia text files.

Write a Python function `construct_positions(docs)` in the file `construct_positions.py`. This function takes the dictionary produced by last week's `construct_database()`.

The function `construct_positions()` should return an appropriate data structure for answering phrase queries (part B). This datastructure should make it efficient to look up for a given word in which document(s) it occurs, and in which position(s) in that document.

You should consider a suitable data-structure, taking into account the following.

Exercises

part B: Phrase queries

Write a Python program `query_phrase.py` which uses the result of part A. As before, the program takes a command line argument which indicates the text file to use. After preprocessing, the program reads lines from standard input. Each line is a query which consists of several words (separated by white space). For each line of input, your program should print all documents in which these words occur as a phrase.

For this week, you should submit (in a single zip file called `w2.zip`) the `construct_database.py`, `construct_positions.py` and `query_phrase.py` scripts. If these scripts use any other scripts, then these should be included as well. I will run your program on a different data-set, so I will use your `construct_database.py`, `construct_positions.py` to create new database(s), and use your `query_phrase.py` for testing.

Exercises

My test script contains UNIX commands such as the following:

```
unzip w2.zip

pycodestyle *.py

echo "op zoek naar" |python3 query_phrase.py small.txt

cat queries.txt | python3 query_phrase.py nlwiki.txt | wc -l
```