CMPE 493
INTRODUCTION TO
INFORMATION RETRIEVAL

Index Compression

Department of Computer Engineering, Boğaziçi University
November 23-24, 2020

Why compression (in general)?

- Use less disk space
  - Saves a little money
- Keep more stuff in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - [read compressed data | decompress] is faster than [read uncompressed data]
  - Premise: Decompression algorithms are fast
    - True of the decompression algorithms we use

Why compression for inverted indexes?

- Dictionary
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - Large search engines keep a significant part of the postings in memory.
    - Compression lets you keep more in memory
- We will devise various IR-specific compression schemes

RCV1: Our collection for this lecture

- Shakespeare's collected works definitely are not large enough for demonstrating many of the points in this course.
- The collection we will use isn't really large enough either, but it is publicly available and is at least a more plausible example.
- As an example for applying scalable index compression/construction algorithms, we will use the Reuters RCV1 collection.
- This is one year of Reuters newswire (part of 1995 and 1996)

A Reuters RCV1 document

REUTERS

You are here: Home > News > Science > Article

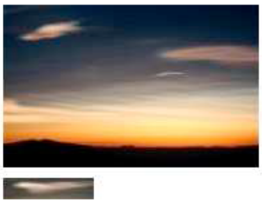
Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]



SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Reuters RCV1 statistics

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= word types)	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
T	tokens	100,000,000

Effect of preprocessing (RCV1 corpus)

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	∆%	cumul %	Size (K)	∆ %	cumul %	Size (K)	∆ %	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52

Lossless vs. lossy compression

- Lossy compression: Discard some information
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
- Lossless compression: All information is preserved.
  - Today's topic

Statistical properties of terms

Statistical Properties of Text

- How fast does vocabulary size grow with the size of a corpus?
- How is the frequency of different words distributed?
- Such factors affect the performance of information retrieval and can be used to select appropriate term weights and other aspects of an IR system.

Vocabulary vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really.
- In practice, the vocabulary will keep growing with the collection size

Vocabulary vs. collection size

- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection
- Typical values (for English): $30 \le k \le 100$ and $b \approx 0.5$
- In a log-log plot of vocabulary size M vs. T, Heaps' law predicts a line with slope about ½
  - log M = log k + b*log T
  - An empirical finding ("empirical law")

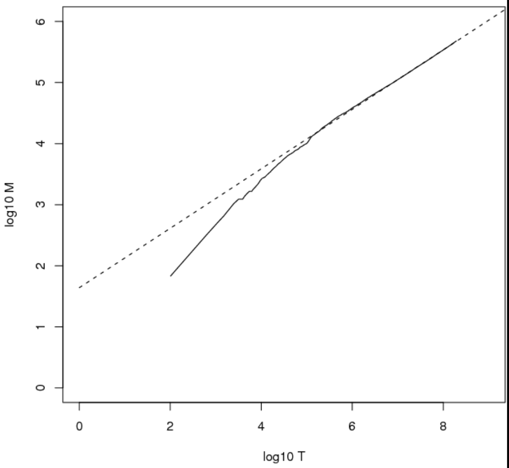
Heaps' Law

For RCV1, the dashed line

Good empirical fit for Reuters RCV1 !

For first 1,000,020 tokens,

- law predicts 38,323 terms;
- actually, 38,365 terms



Word distributions

- Words are not distributed evenly!
- Same goes for letters of the alphabet (ETAOIN SHRDLU), city sizes, wealth, etc.
- Usually, the 80/20 rule applies (80% of the wealth goes to 20% of the people or it takes 80% of the effort to build the easier 20% of the system)…

Shakespeare

- Romeo and Juliet:
  - And, 667; The, 661; I, 570; To, 515; A, 447; Of, 382; My, 356; Is, 343; That, 343; In, 314; You, 289; Thou, 277; Me, 262; Not, 257; With, 234; It, 224; For, 223; This, 215; Be, 207; But, 181; Thy, 167; What, 163; O, 160; As, 156; Her, 150; Will, 147; So, 145; Thee, 139; Love, 135; His, 128; Have, 127; He, 120; Romeo, 115; By, 114; She, 114; Shall, 107; Your, 103; No, 102; Come, 96; Him, 96; All, 92; Do, 89; From, 86; Then, 83; Good, 82; Now, 82; Here, 80; If, 80; An, 78; Go, 76; On, 76; I'll, 71; Death, 69; Night, 68; Are, 67; More, 67; We, 66; At, 65; Man, 65; Or, 65; There, 64; Hath, 63; Which, 60;
  - …
  - A-bed, 1; A-bleeding, 1; A-weary, 1; Abate, 1; Abbey, 1; Abhorred, 1; Abhors, 1; Aboard, 1; Abound'st, 1; Abroach, 1; Absolved, 1; Abuse, 1; Abused, 1; Abuses, 1; Accents, 1; Access, 1; Accident, 1; Accidents, 1; According, 1; Accursed, 1; Accustom'd, 1; Ache, 1; Aches, 1; Aching, 1; Acknowledge, 1; Acquaint, 1; Acquaintance, 1; Acted, 1; Acting, 1; Action, 1; Acts, 1; Adam, 1; Add, 1; Added, 1; Adding, 1; Addle, 1; Adjacent, 1; Admired, 1; Ado, 1; Advance, 1; Adversary, 1; Adversity's, 1; Advise, 1; Afeard, 1; Affecting, 1; Afflicted, 1; Affliction, 1; Affords, 1; Affray, 1; Affright, 1; Afire, 1; Agate-stone, 1; Agile, 1; Agree, 1; Agrees, 1; Aim'd, 1; Alderman, 1; All-cheering, 1; All-seeing, 1; Alla, 1; Alliance, 1; Alligator, 1; Allow, 1; Ally, 1; Although, 1;

The BNC (Adam Kilgarriff)

- 1 6187267 the det
- 2 4239632 be v
- 3 3093444 of prep
- 4 2687863 and conj
- 5 2186369 a det
- 6 1924315 in prep
- 7 1620850 to infinitive-marker
- 8 1375636 have v
- 9 1090186 it pron
- 10 1039323 to prep
- 11 887877 for prep
- 12 884599 i pron
- 13 760399 that conj
- 14 695498 you pron
- 15 681255 he pron
- 16 680739 on prep
- 17 675027 with prep
- 18 559596 do v
- 19 534162 at prep
- 20 517171 by prep

The British National Corpus (BNC) is a 100 million word collection of samples of written and spoken English language from a wide range of sources.

Kilgarriff, A. Putting Frequencies in the Dictionary. *International Journal of Lexicography* 10 (2) 1997. Pp 135--155

Stop words

- 250-300 most common words in English account for 50% or more of a given text.
- Example: "the" and "of" represent 10% of tokens. "and", "to", "a", and "in" - another 10%. Next 12 words - another 10%.
- Moby Dick Ch.1: 859 unique words (types), 2256 word occurrences (tokens). Top 65 types cover 1132 tokens (> 50%).

Zipf's law

- Heaps' law estimates the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law: The *i*th most frequent term has frequency proportional to $1/i$.
- $cf_i \propto 1/i = K/i$ where K is a normalizing constant
- cf_i is <u>collection frequency</u>: the number of occurrences of the term t_i in the collection.

Zipf consequences

- If the most frequent term (*the*) occurs cf_1 times
  - then the second most frequent term (*of*) occurs $cf_1/2$ times
  - the third most frequent term (*and*) occurs $cf_1/3$ times …
- Equivalent: $cf_i = K/i$ where K is a normalizing factor, so
  - $\log cf_i = \log K - \log i$
  - Linear relationship between $\log cf_i$ and $\log i$
- Another power law relationship (like Heaps' Law)

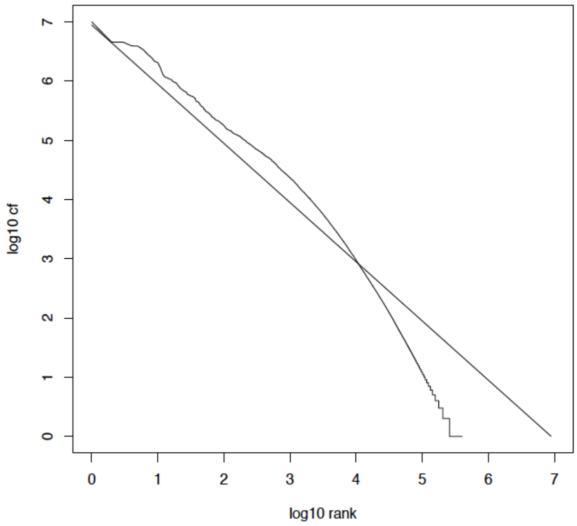
Does Real Data Fit Zipf's Law?

- A law of the form $y = kx^c$ is called a power law.
- Zipf's law is a power law with $c = -1$
- On a log-log plot, power laws give a straight line with slope c.

$$\log(y) = \log(kx^c) = \log k + c\log(x)$$

- Zipf is quite accurate except for very high and low rank.

Zipf's law for Reuters RCV1



Zipf's Law Impact on IR

- Pros:
  - Stopwords account for a large fraction of text. So, eliminating them considerably reduces inverted-index storage costs.
  - Postings list for most remaining words in the inverted index will be short since they are rare, making retrieval fast.
- Cons:
  - Most words very rare. So, gathering sufficient data for meaningful statistical analysis (e.g. for spelling correction) is difficult.

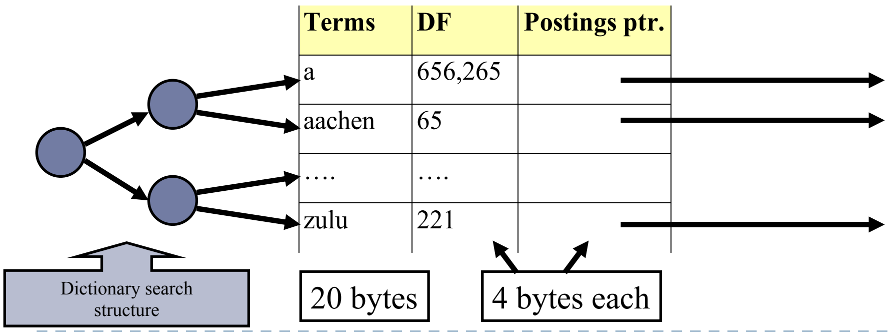
DICTIONARY COMPRESSION

Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important

Dictionary storage - first cut

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.



Terms	DF	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*.
- Average dictionary word in English: ~8 characters

Compressing the term list: Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

DF	Postings ptr.	Term ptr.
33		
29		
44		
126		

Total string length = 400K x 8B = 3.2MB

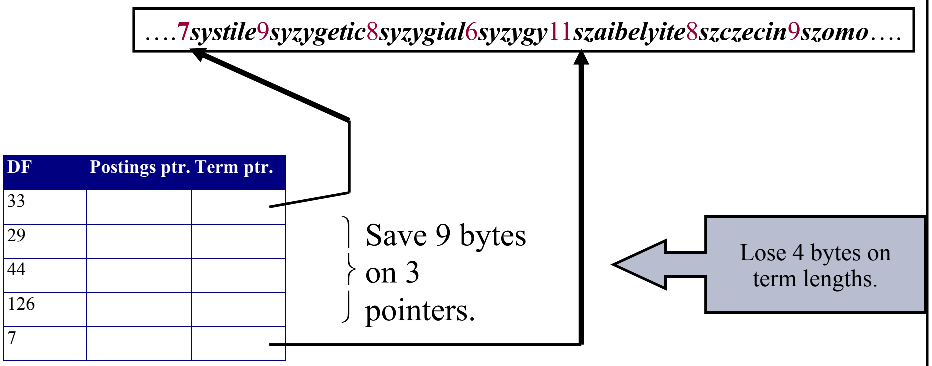
Pointers resolve 3.2M positions: $\log_2 3.2M$ = 22bits = 3bytes

Space for dictionary as a string

- 4 bytes per term for Document Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 400K terms x 19 $\Rightarrow$ 7.6 MB (against 11.2MB for fixed width)

Blocking

- Store pointers to every *k*th term string.
  - Example below: *k*=4.
- Need to store term lengths (1 extra byte)



Net

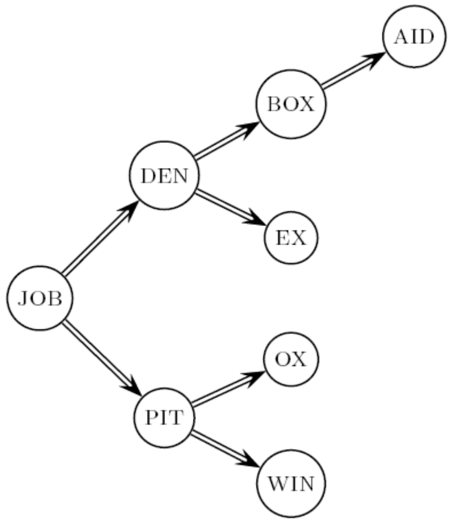
- Example for block size *k* = 4
- Save 5 bytes per four-term block.
- Total: 400,000/4 * 5 = 0.5 MB

Saved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
We can save more with larger *k*.

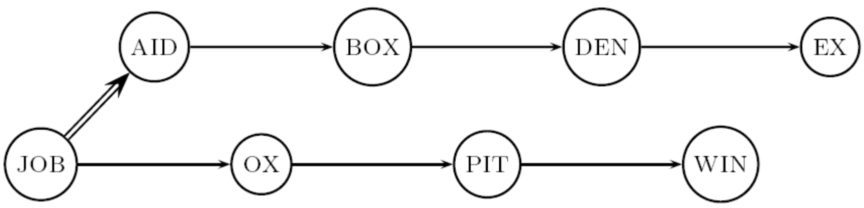
Why not go with larger *k*?

Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = (1+2·2+4·3+4)/8 = ~2.6



Dictionary search with blocking



- Binary search down to 4-term block;
  - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. = (1+2·2+2·3+2·4+5)/8 = 3 comparisons

Front coding

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only

8*automata*8*automate*9*automatic*10*automation*

→8*automat***a*1◊*e*2◊*ic*3◊*ion*

Encodes ***automat***

Extra length beyond ***automat.***

RCV1 dictionary compression summary

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
Also, blocking *k* = 4	7.1
Also, Blocking + front coding	5.9

Character Representations: Fixed length codes

- Binary representations
  - ASCII
  - Representational power (2^k symbols where k is the number of bits)

Most frequent letters in English

- Some are more frequently used than others…
- Most frequent letters:
  - E T A O I N S H R D L U
- Demo:
  - http://www.amstat.org/publications/jse/secure/v7n2/count-char.cfm
- Also: bigrams:
  - TH HE IN ER AN RE ND AT ON NT

Variable length codes

- Alphabet:

```
A .-          N -.        0 -----
B -...        O ---       1 .----
C -.-.        P .--.      2 ..---
D -..         Q --.-      3 ...--
E .           R .-.       4 ....-
F ..-.        S ...       5 .....
G --.         T -         6 -....
H ....        U ..-       7 --...
I ..          V ...-      8 ---..
J .---        W .--       9 ----.
K -.-         X -..-
L .-..        Y -.--
M --          Z --..
```

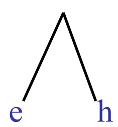
- Demo:
  - http://www.scphillips.com/morse/

Huffman coding

- Developed by David Huffman (1952)
- Average of 5 bits per character (37.5% compression compared to 8 bits)
- Based on frequency distributions of symbols
- Algorithm: iteratively build a tree of symbols starting with the two least frequent symbols

Symbol	*Frequency*
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8

The Huffman tree

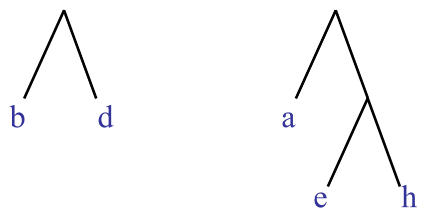


Symbol	Frequency
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5

The Huffman tree

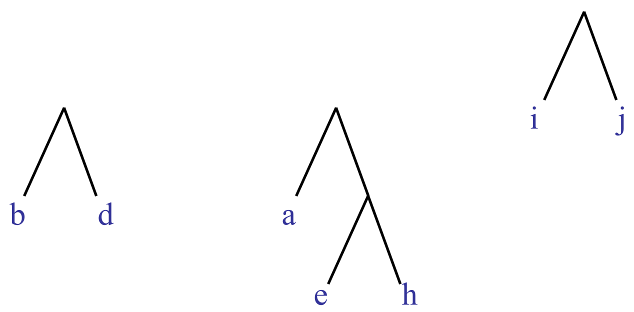
Symbol	*Frequency*
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5
BD	9

The Huffman tree



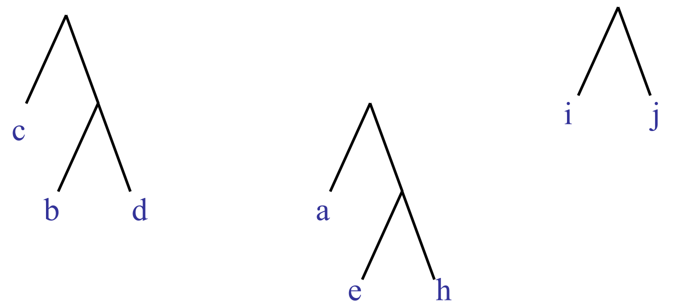
Symbol	*Frequency*
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5
BD	9
AEH	12

The Huffman tree



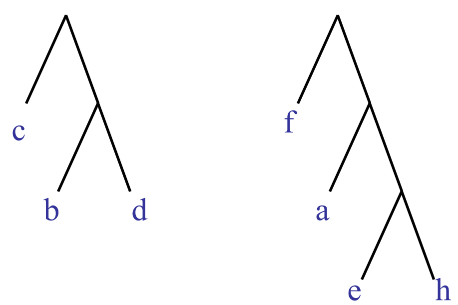
Symbol	*Frequency*
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5
BD	9
AEH	12
IJ	15

The Huffman tree



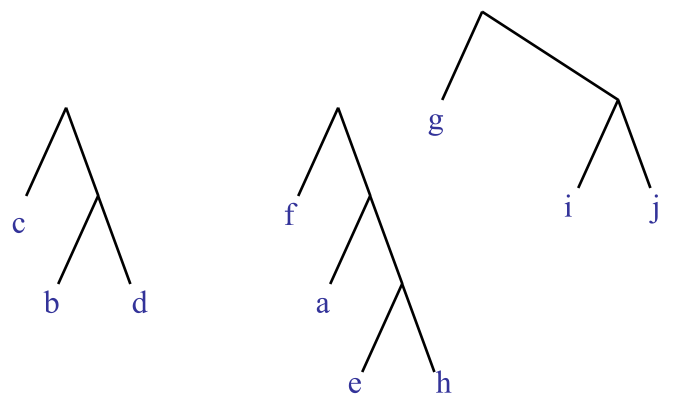
Symbol	*Frequency*
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5
BD	9
AEH	12
IJ	15
CBD	19

The Huffman tree



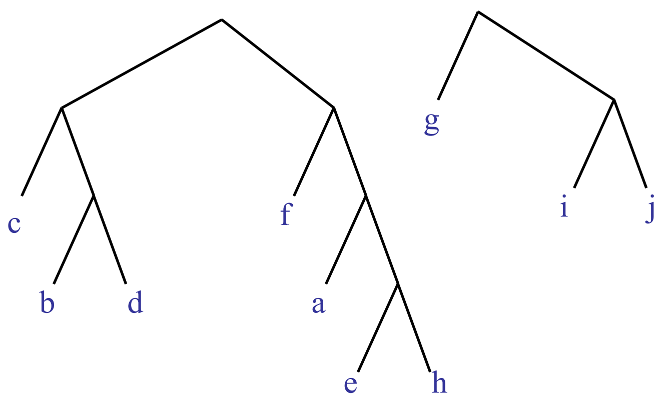
Symbol	*Frequency*
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5
BD	9
AEH	12
IJ	15
CBD	19
FAEH	23

The Huffman tree



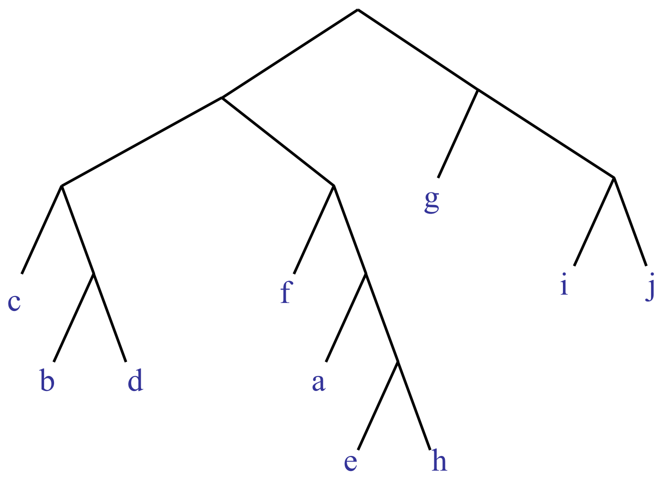
Symbol	***Frequency***
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5
BD	9
AEH	12
IJ	15
CBD	19
FAEH	23
GIJ	30

The Huffman tree

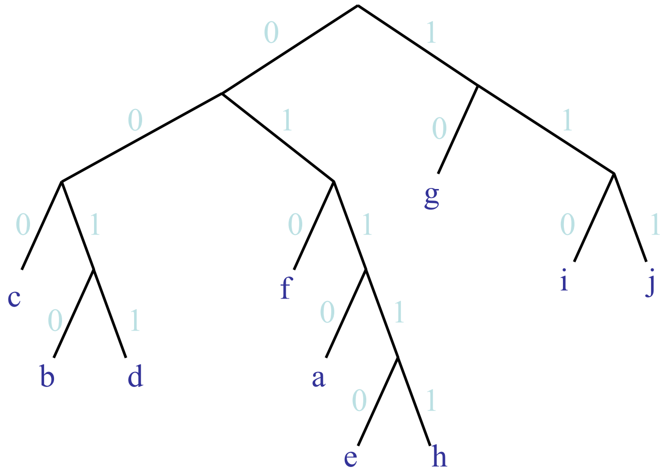


Symbol	Frequency
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8
EH	5
BD	9
AEH	12
IJ	15
CBD	19
FAEH	23
GIJ	30
CBDFAEH	43

The Huffman tree



The Huffman tree



Symbol	*Code*
A	0110
B	0010
C	000
D	0011
E	01110
F	010
G	10
H	01111
I	110
J	111

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800{,}000 \approx 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.

Postings: two conflicting forces

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2$ 1M ~ 20 bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case

Postings file entry

- We store the list of docs containing a term in increasing order of docID.
  - ***computer***: 33,47,154,159,202 …
- Consequence: it suffices to store *gaps*.
  - 33,14,107,5,43 …
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.

Three postings entries

	encoding	postings list									
THE	docIDs	. . .		283042		283043		283044		283045	. . .
	gaps				1		1		1		. . .
COMPUTER	docIDs	. . .		283047		283154		283159		283202	. . .
	gaps				107		5		43		. . .
ARACHNOCENTRIC	docIDs	252000		500100							
	gaps	252000	248100								

Variable length encoding

- Aim:
  - For ***arachnocentric***, we will use ~20 bits/gap entry.
  - For ***the***, we will use ~1 bit/gap entry.
- If the average gap for a term is G, we want to use ~$\log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a *variable length encoding*
- Variable length codes achieve this by using short codes for small numbers

Variable Byte (VB) codes

- For a gap value *G*, we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store *G* and dedicate 1 bit in it to be a <u>continuation</u> bit *c*
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode *G*'s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$.

Example

Binary representation: 1100111000

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings stored as the byte concatenation
000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

Other variable unit codes

- Instead of bytes, we can also use a different "unit of alignment": 32 bits (words), 16 bits, 4 bits (nibbles).
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.
- Variable byte codes: Used by many commercial/research systems

Unary code

- Represent *n* as *n* 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is

11111111111111111111111111111111111111110 .

- Unary code for 80 is:

111111111111111111111111111111111111111111111111
111111111111111111111111111111110

- This doesn't look promising, but….

68

Gamma codes

- We can compress better with <u>bit-level</u> codes
  - The Gamma code is the best known of these.
- Represent a gap *G* as a pair: *length* and *offset*
- *offset* is *G* in binary, with the leading bit cut off
  - For example 13 → 1101 → 101
- *length* is the length of offset
  - For 13 (offset 101), this is 3.
- We encode *length* with *unary code*: 1110.
- Gamma code of 13 is the concatenation of *length* and *offset*: 1110101

Gamma code examples

number	length	offset	γ-code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

Gamma code properties

- G is encoded using $2\lfloor \log G \rfloor + 1$ bits
  - Length of offset is $\lfloor \log G \rfloor$ bits
  - Length of length is $\lfloor \log G \rfloor + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB

Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
  - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

RCV1 compression

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, k = 4	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ–encoded	101.0

Index compression summary

- We can now create an index for highly efficient retrieval that is very space efficient
- Only 4% of the total size of the collection
- Only 10-15% of the total size of the text in the collection
- However, we've ignored positional information
- Hence, space savings are less for indexes used in practice
  - But techniques substantially the same.

References

- *Introduction to Information Retrieval*, chapter 5.
  - http://nlp.stanford.edu/IR-book/information-retrieval-book.html
- The slides were adapted from the lecture notes at the book's website and Prof. Dragomir Radev' s and Prof. Raymond Mooneys's lecture notes.