COM3110/4115/6115:

Text Processing

Text Compression

Rob Gaizauskas

Department of Computer Science
University of Sheffield

Overview

- Models
  - Static
  - Semi-static
  - Adaptive
- Coding
  - Huffman Coding
  - Arithmetic Coding
- Further topics:
  - Symbolwise Models
  - Dictionary Methods
  - Synchronisation
  - Performance Issues

Introduction

- Have seen a dramatic increase of
  - low cost disk storage
  - transmission bandwith
  - processor speed
- But also a massive increase in data volume:
  - text, sound and images
  - so, techniques to *compress* data remain significant
- We shall concentrate on techniques for *text compression*
- Text compression distinct from some other forms of data compression:
  - the text must be *exactly reconstructable*
  - not so critical for digitised analogue signals, such as image/sound
  - text compressions requires so-called *lossless coding*

Introduction (ctd)

- Distinguish *lossless* vs. *lossy* compression methods
- Lossless Compression:
  - ◇ class of algorithms allowing original data to be perfectly reconstructed from compressed data
- Lossy Compression:
  - ◇ achieve data reduction by *discarding* (i.e. losing) information
  - ◇ suitable for certain media types, esp.: image / video / audio data
    - widely used in data *streaming* contexts

    e.g. achieve data reduction of an image by computing a version with lower pixel density
  - ◇ Text data requires *lossless* compression
    - "text from which N% of info discarded" doesn't make sense
    - expect decompression to return text identical to original in form/content

(See Wikipedia pages for Lossy and Lossless Compression)

Introduction (ctd)

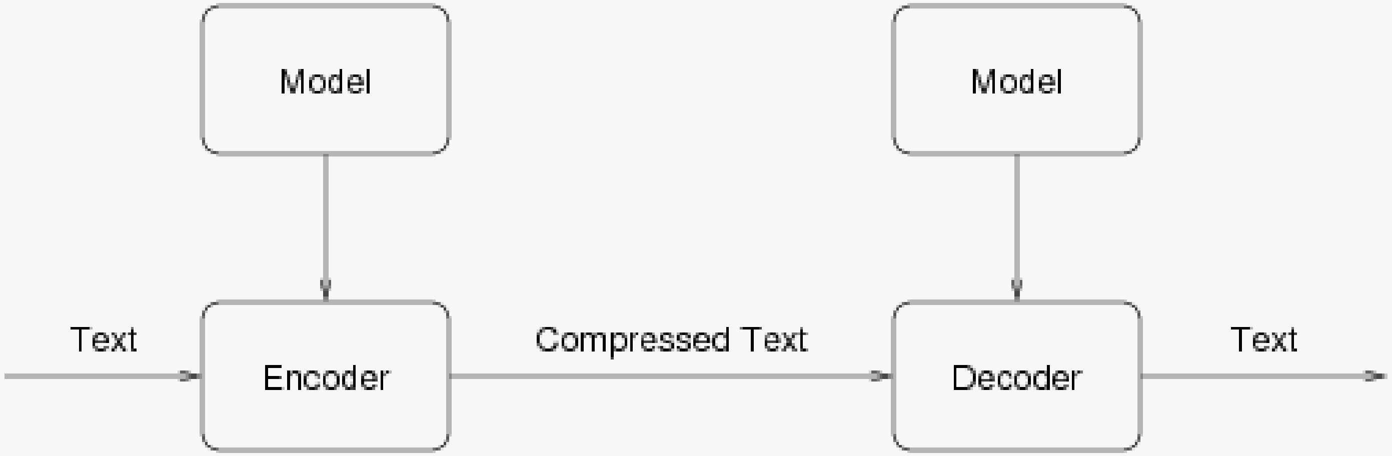
- Text compression techniques may be classified in several ways.
  - *symbolwise* vs. *dictionary* methods
  - *static* vs. *adaptive* methods
- Dictionary methods work by replacing word/text fragments with an index to an entry in a dictionary
- Symbolwise methods work by estimating the probabilities of symbols (characters/words) and coding one symbol at a time using shorter codewords for the more likely symbols
  - rely on a **modeling** step and a **coding** step
  - *modeling*: estimation of probabilities for the symbols in the text
    - the better the probability estimates, the higher the compression that can be achieved
  - *coding*: conversion of probabilities from model into a bitstream

Introduction (ctd)

- **Static** vs. **adaptive** methods:
  - *Static*: use a fixed model or fixed dictionary derived in advance of any text to be compressed
  - *Semi-static*: use current text to build a model or dictionary during one pass, then apply it in second pass
  - *Adaptive*: build model or dictionary adaptively during one pass

Coding and Decoding

- Function of model is to predict symbols
  - model amounts to a probability distribution for all possible symbols, i.e. the "alphabet"
- The encoder uses the model to encode (compress) the text
- The decoder must use the same model to decode it



Information Content

- The number of *bits* in which a symbol s should be coded is its *information content*, denoted $I(s)$
- Information content related to *predicted probability* $P[s]$, as follows:

$$I(s) = -log_2 P[s]$$

e.g. for a fair coin toss in which the outcome is "heads", the best an encoder can do it use $-log_2(\frac{1}{2}) = 1$ bit.

- The *entropy* H of the probability distribution — the average information per symbol over the whole alphabet — is given by:

$$H = \sum_s P[s] \cdot I(s) = \sum_s -P[s] \cdot log_2 P[s] \quad \text{bits/character}$$

  - ◇ akin to 'average' of code lengths *weighted* by probability

- The entropy H places a *lower bound* on compression
  - ◇ Shannon's Source Coding Theorem

Information Content (ctd)

Examples:

- With a 'fair' coin, $P(\text{head}) = P(\text{tail}) = 0.5$
  - hence: $I(\text{head}) = I(\text{tail}) = -log_2(0.5) = 1$
  - hence: $H = 0.5 \cdot 1 + 0.5 \cdot 1 = 1$
- With a 'biased' coin, such that $P(\text{head}) = 0.99, \quad P(\text{tail}) = 0.01$
  - $I(\text{head}) = -log_2(0.99) = 0.0145, \qquad I(\text{tail}) = -log_2(0.01) = 6.644$
  - $H = 0.99 \cdot 0.0145 + 0.01 \cdot 6.644 = 0.0808$
- For a 'fair' 8-sided dice, $P(s) = \frac{1}{8} = 0.125$ for each side s
  - $I(s) = -log_2(\frac{1}{8}) = 3$, and hence $H = 3$ also
- For a 'baised' 8-sided dice, with $P(s_0) = 0.9$ and other $P(s_i) = 0.0143$
  - $I(s_0) = -log_2(0.9) = 0.152, \quad I(s_i) = -log_2(0.0143) = 6.13$
  - $H = 0.7498$

Models and Context

- Probability of encountering a given symbol at a particular place in a text is influenced by *preceding symbols*

  e.g. probability of *u* following *q* much higher than *u* occurring on average

- Models that take immediately preceding symbols into account are called finite-context models

  - best text compression results consider contexts of 3-5 characters

- Models that *ignore* preceding content are called zero order models

Static Zero Order Character-based Model for Moby Dick

Ch	Count	Pr	Ch	Count	Pr	Ch	Count	Pr	Ch	Count	Pr
SP	198111	0.1623	p	16207	0.0132	C	1148	0.0009	K	178	0.0001
e	115863	0.0949	b	15453	...	P	1049	...	V	171	...
t	85544	0.0701	v	8428		x	1008		1	140	
a	75267	...	k	7882		?	1004		0	131	
o	68341		.	7558		O	990		2	60	
n	64434		-	5984		L	901		8	58	
s	62023		;	4174		j	830		5	54	
i	61893		I	3544		R	824		7	53	
h	61435		”	3071		F	804		3	47	
r	751314		’	2922		M	754		*	45	
l	741896		A	2651		D	752		4	39	
d	37469		T	2458		G	641		Z	38	
u	26458		S	2209		z	594		6	37	
NL	22933		!	1767		Y	331		9	35	
m	22526		H	1465		Q	322		_	26	
c	21361		B	1427		J	253		X	23	
w	20917		W	1306		U	240		&	2	
g	20181		E	1240		(	215		$	2	
f	20031		q	1234		)	215		[	2	
,	19230		N	1186		:	196		]	2	1.6e-06
y	16543										

- 81 characters in alphabet
- total character occurrences = 1,220,150
- entropy using this model: 4.4953 bits/char

Static vs. Adaptive Modelling

Probabilities may be estimated in various ways

- **Static modelling** derives, and then uses, a single model for all texts
  - ◇ will perform poorly on texts different from those used in constructing the model, e.g. texts with tables of numbers
- **Semi-static modelling** derives model for the file in a 1st pass
  - ◇ model derived will be better suited to the text than a static one
  - ◇ but, is inefficient because:
    - — must make two passes over text
    - — must also transmit model
- **Adaptive modelling** derives model during encoding

Adaptive Models

- Adaptive models begin with a *base* probability distribution
  - refine the model as more symbols are encountered, during encoding
  - so, the text being encoded itself *re-defines the model*
- Decoder starts with same base probability distribution
  - it is decoding the *same symbol sequence*
  - so, it can refine the model in the same way
- *Issues*:
  - Care must be taken to ensure no character is ever predicted with zero probability simply because it has not been seen yet
  - Principal disadvantage: not suitable for random access to files
    - a text can only be decoded from the beginning
    - poses difficulties for retrieval applications

Adaptive Models (ctd)

- *EXAMPLE*: encoding Moby Dick
  - ◇ might start assuming a uniform probability of $1/81$ per char
    - implies same minimal possible code length for all symbols at $-log_2(1/81) = 6.34$ bits
  - ◇ after encoding (say) 100,000 chars, and observing char distributions, we have a much more accurate 0-order model
  - ◇ suppose now about to encode `e` of `whal`**`e`**:
    - by now have observed that ∼10% of chars are `e`
    - thus, the `e` can be encoded in $\sim -log_2 0.1 = 3.32$ bits
    - decoder will refine its model in parallel manner for use in decoding

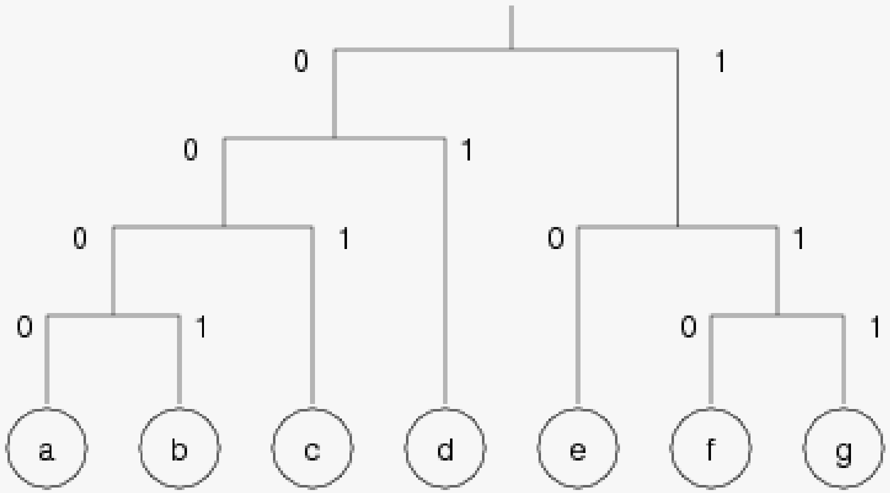
Huffman Coding

- **Coding** is the task of determining the output representation of a symbol, given the probability distribution supplied by the model
- Coder should output:
  - short codes for high probability symbols
  - long codes for low probability symbols
- Speed of coder may also be significant
  - computing optimal codes can be slow
  - hence, there is a trade-off between speed of compression and compression rate
- Huffman coding dates from 1952
  - was the dominant model till the 1970's
  - with refinements, it still has applications today, e.g. in text retrieval

Huffman Coding (ctd)

- Technique uses a **code tree** for encoding and decoding
  - ◇ each branch is labelled with a 0 or 1
  - ◇ each leaf node is a symbol in the alphabet
  - ◇ code is **prefix-free** – no codeword is the prefix of another
- Consider a seven symbol alphabet (example from Witten *et al.*)

Symbol	Prob	Codeword
a	0.05	0000
b	0.05	0001
c	0.1	001
d	0.2	01
e	0.3	10
f	0.2	110
g	0.1	111



e.g. eefggfed is coded as 10101101111111101001

Huffman Coding (ctd)

- Given a set of codewords produced by the algorithm, can compute the expected *average* code length as follows:
  - multiply each symbol code length by associated probability
  - sum results across all symbols

Symbol	Prob	Code	*len*	$p \times len$
a	0.05	0000	4	0.2
b	0.05	0001	4	0.2
c	0.1	001	3	0.3
d	0.2	01	2	0.4
e	0.3	10	2	0.6
f	0.2	110	3	0.6
g	0.1	111	3	0.3
				2.6

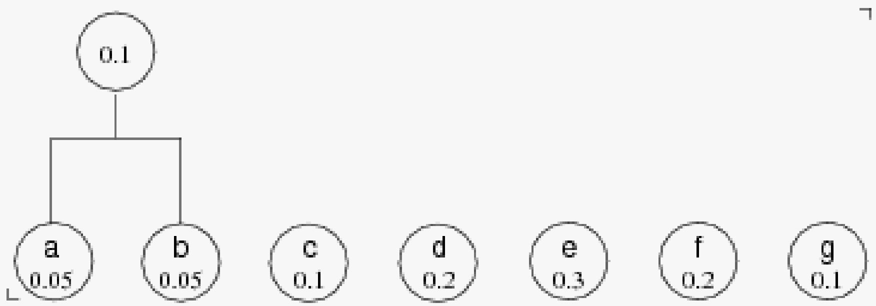
- Compare this to minimal *fixed-length* code length for same symbol set

  e.g. for above symbol set, could use a 3-bit fixed length code
  - 3-bit fixed length code allows for $2^3 = 8$ symbols

Huffman — the coding algorithm

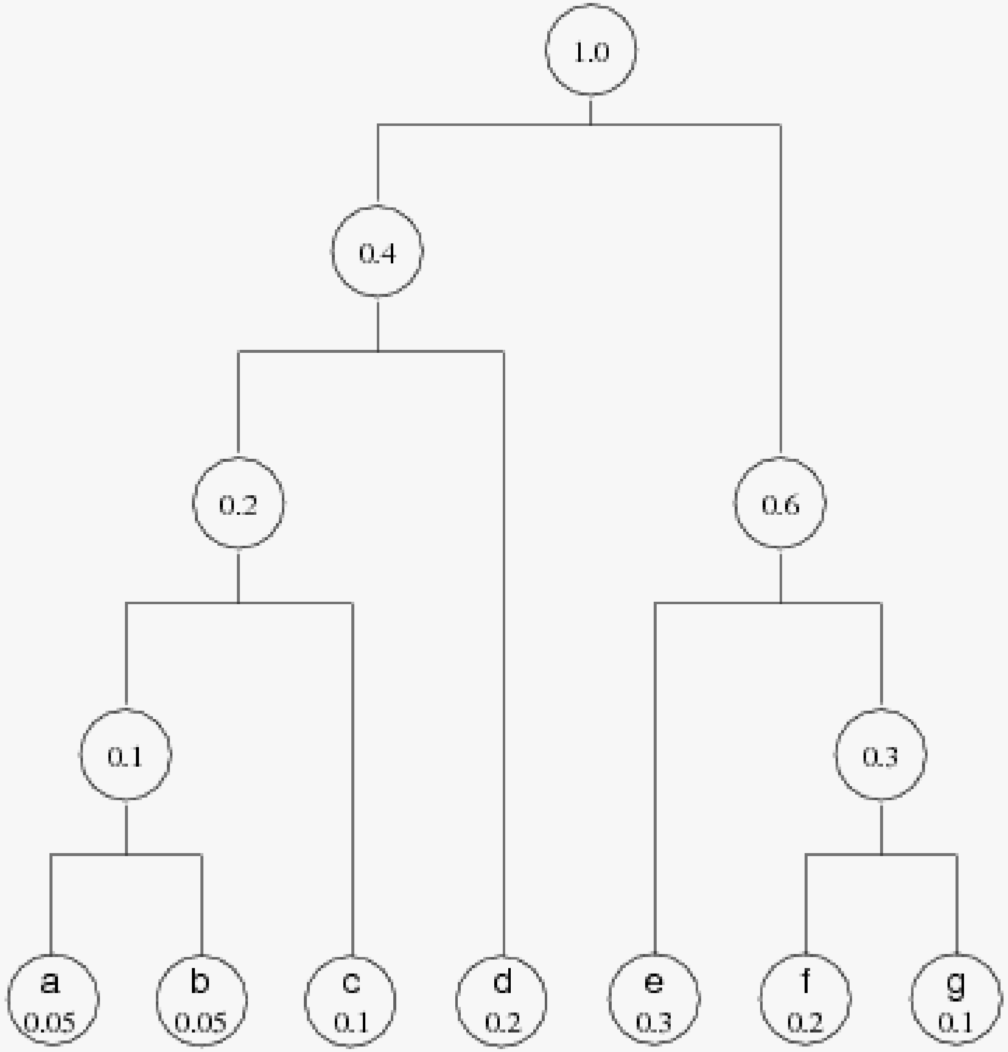
- The code tree is constructed bottom up from the probabilistic model according to the following algorithm:
  (a) Probabilities are associated with leaf nodes
  (b) Identify the two nodes with smallest probabilities
    - join them under a parent node, whose probability is their sum
  (c) Repeat step (b) until only one node remains
  (d) 0's and 1's are then assigned to each binary split
- *EXAMPLE*:
  ◇ start with leaf nodes:

  ◇ join two nodes with smallest probabilities:



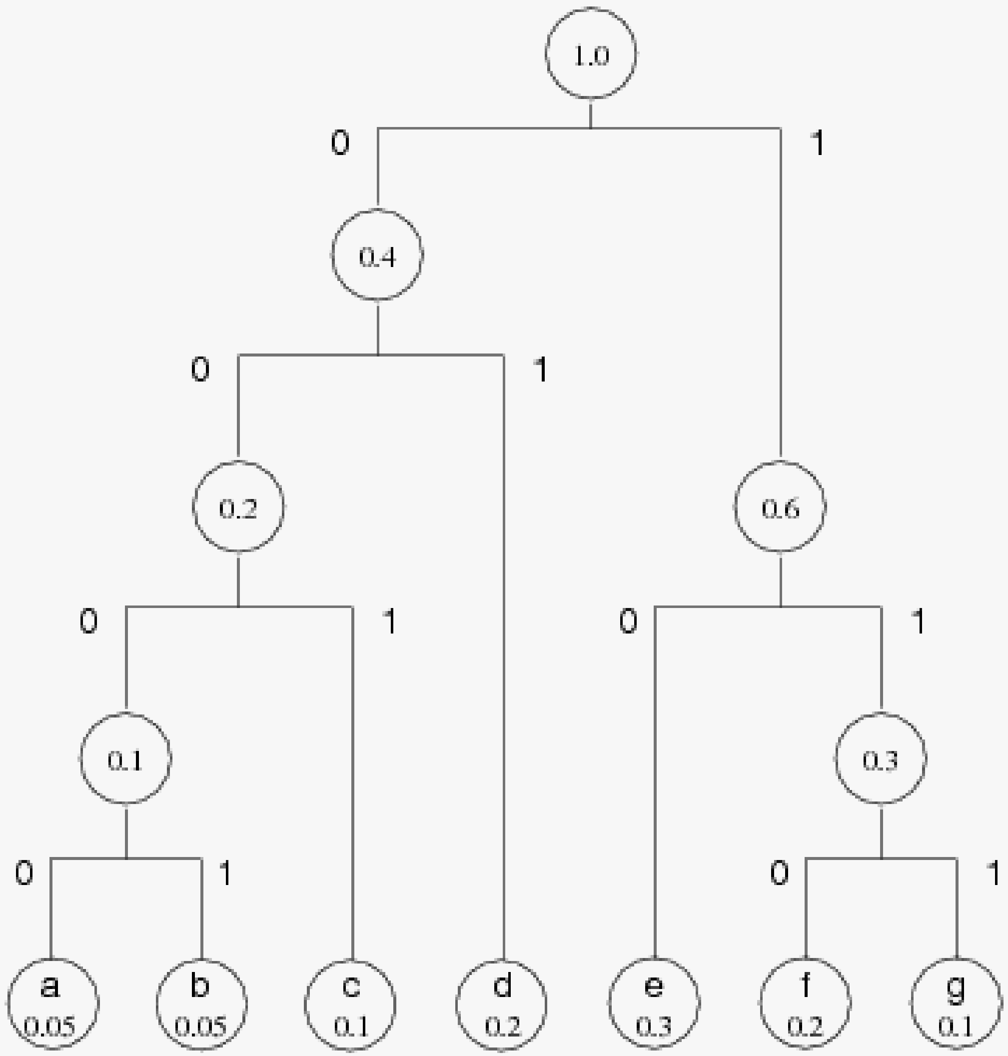
Huffman — the coding algorithm (ctd)

- repeat until only one node remains . . .



Huffman — the coding algorithm (ctd)

- assign 0/1 to branches of each binary fork
  - doesn't matter which is which



Huffman Coding — further details

- Huffman coding is fast for both encoding and decoding
  - provided probability distribution is static
- Variants of Huffman coding developed for adaptive models
  - *but* these are complex
  - generally better to use arithmetic coding for adaptive models
- Huffman effective when used with a word-based model, rather than a character-based model
  - gives good compression
  - fast
  - supports random access to compressed files
    (given some additional requirements on how method used)

Canonical Huffman codes

- Huffman gives effective compression where there are *many symbols*, with a highly *skewed* distribution
  - ◇ arises, e.g. for word-based models
  - ◇ i.e. many symbols (words), some v.frequent, some rare
- Tree-based storage of *v.large models* is *costly & inefficient*
  - ◇ tree nodes store pointers to child nodes — *costly for memory*
  - ◇ *traversing* tree involves much *jumping* between locations — *inefficient*
- These issues addressed by use of canonical Huffman codes
  - ◇ special Huffman code, where codes generated in a *standardised format*
  - ◇ specifically, all codes for given code-length *assigned values sequentially*
  - ◇ this feature allows for both
    - efficient storage and/or transmission of codebook
    - more efficient decoding algorithm

Canonical Huffman codes (ctd)

- To create a canonical code:
  - first determine length of code for each symbol
    - can do this by applying standard Huffman coding algorithm
  - group symbols having same code-length into blocks, and order
    - e.g. could sort alphabetically, or in some other way
  - assign codes by '*counting up*' – addressing blocks in code-length order
- To illustrate, consider our 7-letter example again:

Symbol	Prob	Code	*len*
a	0.05	0000	4
b	0.05	0001	4
c	0.1	001	3
d	0.2	01	2
e	0.3	10	2
f	0.2	110	3
g	0.1	111	3

  - ignore original codes, and group symbols by their code-length, i.e.:

```
[2] d  e   [3] c  f  g   [4] a  b
```

Canonical Huffman codes (ctd)

- *Example – continued*: creating a canonical code ...
  - First, group symbols by their code lengths:

    `[2] d  e  [3] c  f  g  [4] a  b`

  - Assign codes:
    - assign first symbol a 'zero' code of required length
    - for successive symbols, simply count up 1
    - if code length goes up, then (i) count up & (ii) add 0s to get new len

```
[2] d  e   [3] c   f   g   [4] a    b
    00 01      100 101 110     1110 1111
```

- To store / transmit this code, is sufficient to specify:
  - sequence of symbols: `d e c f g a b`
  - number of symbols at each code-length: `(0, 2, 3, 2)`
    - i.e. *len 1: 0 items; len 2: 2 items; len 3: 3 items; len 4: 2 items*
  - this is sufficient info to reconstruct entire code

Canonical Huffman codes (ctd)

- This approach also supports an efficient decoding algorithm
  - does not require a code tree
- Method:
  - store blocks of symbols in sequential order
  - compute code for only the *first* symbol in each block
  - by comparing prefixes of input to these first-symbol codes ($<, >$), can determine which block next code belongs to, and hence its length
  - binary difference between next code and first-symbol code gives position of symbol in block sequence
- *Example*:
  - Example on last slide has first-symbol codes: 00, 100, 1110
  - Assume input: 11001111.......
  - Find: prefix 11 $>$ 00; prefix 110 $>$ 100; prefix 1100 $<$ 1110
    - so next codeword is 110, of the length 3 block
  - Difference of 100 and 110 shows symbol is 3rd item in block sequence

Reading

- Main:
  - Baeza-Yates and Ribeiro-Neto, Ch 7.4-7.5
- Other:
  - I. H. Witten, A. Moffat, T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd ed. Morgan Kaufmann. 1999.
  - Nam Phamdo. Theory of Data Compression. `www.data-compression.com/theory.html`