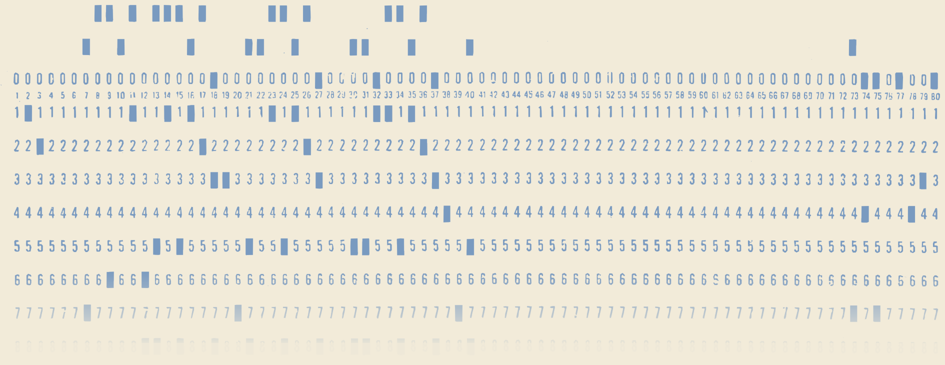


Lecture 13: Data Compression I

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: November 14, 2024

Announcements

1. Programming Assignment 2 posted soon
2. Quiz 5 due Friday
   - Covers string matching
   - 2 questions (multiple choice)
   - Usual rules apply
3. Attendance Code:

Meeting Goals

Discuss data compression!

- Introduce the data compression task
- Define character encoding and related terminology
- Define prefix codes
- Construct Huffman codes
- Prove optimality of Huffmann codes

Data Compression

The Story So Far

Emphasis. How do we process data?

- Data structures
  - How can we organize data perform primitive operations efficiently?
- Fundamental operations on arbitrary data:
  - sorting
  - string matching

The Story So Far

Emphasis. How do we process data?

- Data structures
  - How can we organize data perform primitive operations efficiently?
- Fundamental operations on arbitrary data:
  - sorting
  - string matching

A New Question. How do we *store* and *transmit* data efficiently?

New Topics. Fundamental problems

1. Data Compression (starting today)
   - how to store data using as little space as possible
2. Error Correction (following topic)
   - how to

The Data Compression Task

Terminology.

- **source text:** string $S \in \Sigma_S^*$ to be stored/transmitted
  - Σ_S is some alphabet, e.g., Roman alphabet
- **coded text:** encoded data $C \in \Sigma_C^*$ that is actually stored/transmitted
  - typically have $\Sigma_C = \{0, 1\}$

The Data Compression Task

Terminology.

- **source text:** string $S \in \Sigma_S^*$ to be stored/transmitted
  - Σ_S is some alphabet, e.g., Roman alphabet
- **coded text:** encoded data $C \in \Sigma_C^*$ that is actually stored/transmitted
  - typically have $\Sigma_C = \{0, 1\}$
- **encoding:** An algorithm E that maps source texts to coded texts
  - $E : \Sigma_S^* \to \Sigma_C^*$
- **decoding:** An algorithm D that maps encoded texts to decoded texts
  - $D : \Sigma_C^* \to \Sigma_S^*$

The Data Compression Task

Terminology.

- **source text:** string $S \in \Sigma_S^*$ to be stored/transmitted
  - Σ_S is some alphabet, e.g., Roman alphabet
- **coded text:** encoded data $C \in \Sigma_C^*$ that is actually stored/transmitted
  - typically have $\Sigma_C = \{0, 1\}$
- **encoding:** An algorithm E that maps source texts to coded texts
  - $E : \Sigma_S^* \to \Sigma_C^*$
- **decoding:** An algorithm D that maps encoded texts to decoded texts
  - $D : \Sigma_C^* \to \Sigma_S^*$

Goal. Represent S using as little **space** as possible.

The Data Compression Task

Terminology.

- **source text:** string $S \in \Sigma_S^*$ to be stored/transmitted
  - Σ_S is some alphabet, e.g., Roman alphabet
- **coded text:** encoded data $C \in \Sigma_C^*$ that is actually stored/transmitted
  - typically have $\Sigma_C = \{0, 1\}$
- **encoding:** An algorithm E that maps source texts to coded texts
  - $E : \Sigma_S^* \to \Sigma_C^*$
- **decoding:** An algorithm D that maps encoded texts to decoded texts
  - $D : \Sigma_C^* \to \Sigma_S^*$

Lossy vs. Lossless Compression.

- **Lossless Compression.** decoding recovers original text: $D(E(S)) = S$
  - Examples: zip (general archive), flac (audio), tiff (image)

The Data Compression Task

Terminology.

- **source text:** string $S \in \Sigma_S^*$ to be stored/transmitted
  - Σ_S is some alphabet, e.g., Roman alphabet
- **coded text:** encoded data $C \in \Sigma_C^*$ that is actually stored/transmitted
  - typically have $\Sigma_C = \{0, 1\}$
- **encoding:** An algorithm E that maps source texts to coded texts
  - $E : \Sigma_S^* \to \Sigma_C^*$
- **decoding:** An algorithm D that maps encoded texts to decoded texts
  - $D : \Sigma_C^* \to \Sigma_S^*$

Lossy vs. Lossless Compression.

- **Lossless Compression.** decoding recovers original text: $D(E(S)) = S$
  - Examples: zip (general archive), flac (audio), tiff (image)
- **Lossy Compression.** decoding approximates original text: $D(E(S)) \approx S$
  - Examples: mp3 (audio), jpg (image), mpg (video)

The Data Compression Task

Terminology.

- **source text:** string $S \in \Sigma_S^*$ to be stored/transmitted
  - Σ_S is some alphabet, e.g., Roman alphabet
- **coded text:** encoded data $C \in \Sigma_C^*$ that is actually stored/transmitted
  - typically have $\Sigma_C = \{0, 1\}$
- **encoding:** An algorithm E that maps source texts to coded texts
  - $E : \Sigma_S^* \to \Sigma_C^*$
- **decoding:** An algorithm D that maps encoded texts to decoded texts
  - $D : \Sigma_C^* \to \Sigma_S^*$

Lossy vs. Lossless Compression.

- **Lossless Compression.** decoding recovers original text: $D(E(S)) = S$
  - Examples: zip (general archive), flac (audio), tiff (image)
- **Lossy Compression.** decoding approximates original text: $D(E(S)) \approx S$
  - Examples: mp3 (audio), jpg (image), mpg (video)

Our Focus: lossless compression!

The Quality of an Encoding Scheme

Goals of Encoding

- Efficiency of encoding/decoding
- resilience to errors/noise in transmission
- security (encryption)
- integrity (detect modifications)
- size

The Quality of an Encoding Scheme

Goals of Encoding

- Efficiency of encoding/decoding
- resilience to errors/noise in transmission
- security (encryption)
- integrity (detect modifications)
- **size**

Our focus. Minimize the **size** of the encoded text.

- *data compression*

The Quality of an Encoding Scheme

Our focus. Minimize the **size** of the encoded text.

- *data compression*

Measure of quality. The *compression ratio:*

$$\frac{|C| \cdot \log|\Sigma_C|}{\big|S \cdot \log|\Sigma_S|\big|} \overset{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S| \cdot \log|\Sigma_S|}$$

The Quality of an Encoding Scheme

Our focus. Minimize the **size** of the encoded text.

- *data compression*

Measure of quality. The *compression ratio:*

$$\frac{|C| \cdot \log|\Sigma_C|}{\big|S \cdot \log|\Sigma_S|\big|} \overset{\Sigma_C = \{0,1\}}{=} \frac{|C|}{|S| \cdot \log|\Sigma_S|}$$

Question. Why all of the $\log|\Sigma|$s?

The Quality of an Encoding Scheme

Our focus. Minimize the **size** of the encoded text.

- *data compression*

Measure of quality. The *compression ratio:*

$$\frac{|C| \cdot \log|\Sigma_C|}{\left|S \cdot \log|\Sigma_S|\right|} \overset{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S| \cdot \log|\Sigma_S|}$$

Question. Why all of the $\log|\Sigma|$s?

- $\lceil \log \sigma \rceil$ is the minimum number of bits needed to represent σ distinct values (in binary)
- there are 2^b distinct binary strings of length b

The Quality of an Encoding Scheme

Our focus. Minimize the **size** of the encoded text.

- *data compression*

Measure of quality. The *compression ratio:*

$$\frac{|C| \cdot \log|\Sigma_C|}{\left|S \cdot \log|\Sigma_S|\right|} \overset{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S| \cdot \log|\Sigma_S|}$$

Interpretation. Compression ratios:

$< 1 \implies$ compression

- smaller values are better

$= 1 \implies$ no compression

$> 1 \implies$ encoded text is larger(?!)

- this is sometimes unavoidable …foreshadowing to next week

Data Compression Roadmap

Questions. When, how, and how much can we compress?

- **Part I:** Exploiting non-uniform character frequencies
  - Huffman Codes
- **Interlude:** Limits of data compression
- **Part II:** Exploiting repetition in texts
  - Run-length encoding
  - Lempel-Ziv-Welch (LZW) encoding
- **Part III:** Creating repetition in texts
  - Move-to-front transform
  - Burrows-Wheeler transform

Character Encoding

Question. How do computers encoded English language text?

Question. How do computers encoded English language text?

Historical answer. ASCII use 7 bits per character

- all characters treated equally
- $2^7 = 128$ possible characters

				b7 b6 b5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b4 ↓	b3 ↓	b2 ↓	b1 ↓	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	´	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Question. How do computers encoded English language text?

Historical answer. ASCII use 7 bits per character

- all characters treated equally
- $2^7 = 128$ possible characters

Modern answer. Unicode

- $\sim 150{,}000$ representable characters (different scripts, emoji, etc.)
- several encoding schemes character → bits
- different characters' representations can have different lengths
    - e.g., ASCII characters represented by 8 bits

Question. How do computers encoded English language text?

Historical answer. ASCII use 7 bits per character

- all characters treated equally
- $2^7 = 128$ possible characters

Modern answer. Unicode

- $\sim 150,000$ representable characters (different scripts, emoji, etc.)
- several encoding schemes character → bits
- different characters' representations can have different lengths
  - e.g., ASCII characters represented by 8 bits

Character Encoding. Encode each character individually $E : \Sigma_S \to \Sigma_C^*$

- typically, $|\Sigma_S| \gg |\Sigma_C|$ $(= 2)$, so need several bits per character
- for $c \in \Sigma_S$, call $E(c)$ the **codeword** of c
- to encode a text, encode individual characters and concatenate

Question. How do computers encoded English language text?

Historical answer. ASCII use 7 bits per character

- all characters treated equally
- $2^7 = 128$ possible characters

Modern answer. Unicode

- $\sim 150,000$ representable characters (different scripts, emoji, etc.)
- several encoding schemes character $\to$ bits
- different characters' representations can have different lengths
  - e.g., ASCII characters represented by 8 bits

Character Encoding. Encode each character individually $E : \Sigma_S \to \Sigma_C^*$

- typically, $|\Sigma_S| \gg |\Sigma_C|$ $(= 2)$, so need several bits per character
- for $c \in \Sigma_S$, call $E(c)$ the **codeword** of c
- to encode a text, encode individual characters and concatenate

Fixed vs. Variable Length Encoding

- *fixed length encoding* $\implies$ all codewords have the same length (e.g. ASCII)
- *variable length encoding* $\implies$ different lengths for different codewords (e.g. Unicode)

Fixed Length Codes

Advantages of fixed length codes

- fast decoding
  - use a lookup-table
  - can be as fast as a single array access
- local encoding
  - if character length is B, ith character starts at index $i \cdot B$

Fixed Length Codes

Advantages of fixed length codes

- fast decoding
  - use a lookup-table
  - can be as fast as a single array access
- local encoding
  - if character length is B, ith character starts at index $i \cdot B$

Example. For (8-bit) ASCII encoding, how many (Roman alphabet) characters is this text? Where are the character divisions?

`011101000110010101110000 1110100`

Fixed Length Codes

Advantages of fixed length codes

- fast decoding
  - use a lookup-table
  - can be as fast as a single array access
- local encoding
  - if character length is B, ith character starts at index $i \cdot B$

Example. For (8-bit) ASCII encoding, how many (Roman alphabet) characters is this text? Where are the character divisions?

```
011101000110010101110000001110100
```

Disadvantages of fixed length codes

- Inflexible (non-extensible)
  - how can we represent this awesome new emoji???
- Space inefficient
  - infrequently used characters require as much space as common characters
  - common characters are longer than they need to be

Variable Length Codes

Variable Length
Advantages:

- more flexibility
- compressibility?

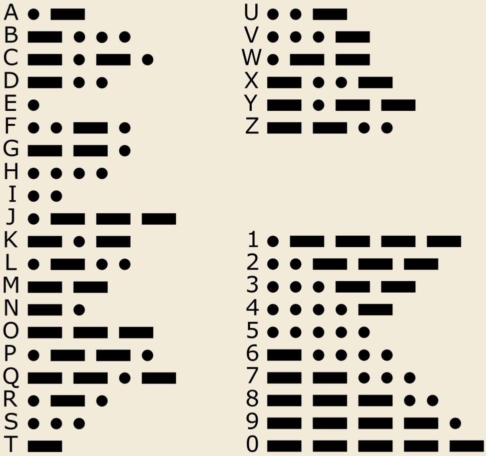
Variable Length Codes

Variable Length Advantages:

- more flexibility
- compressibility?

An old idea. Morse Code

- encode characters as "dots" and "dashes"
- more common characters are shorter



Variable Length Codes

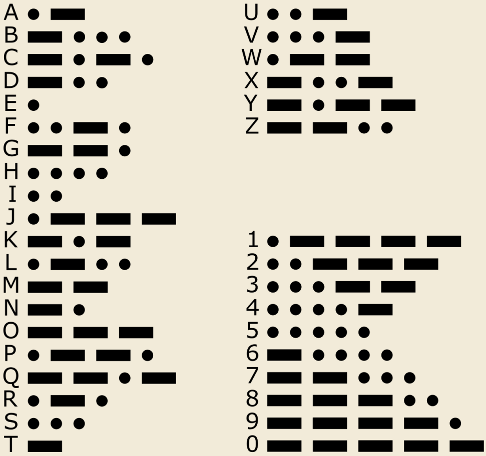
Variable Length Advantages:

- more flexibility
- compressibility?

An old idea. Morse Code

- encode characters as "dots" and "dashes"
- more common characters are shorter

Question. How many characters in the Morse code encoding?



Codes Misbehaving

PollEverywhere

Consider the following code

c	a	n	b	s
$E(c)$	0	10	110	100

What is the original text corresponding to the encoded text 1100100100?

pollev.com/comp526

Codes Misbehaving

Question. What was the issue with this code?

PollEverywhere

Consider the following code

c	a	n	b	s
$E(c)$	0	10	110	100

What is the original text corresponding to the encoded text 1100100100?

pollev.com/comp526

Codes Misbehaving

Question. What was the issue with this code?

- The *relationship* between $E(n) = 10$ and $E(s) = 100$
  - If we read 10 in the encoded text, are we done reading a character?

PollEverywhere

Consider the following code

c	a	n	b	s
$E(c)$	0	10	110	100

What is the original text corresponding to the encoded text 1100100100?

pollev.com/comp526

Codes Misbehaving

Question. What was the issue with this code?

- The *relationship* between $E(n) = 10$ and $E(s) = 100$
  - If we read 10 in the encoded text, are we done reading a character?
- "Reasonable" codes should avoid this ambiguity!
  - We should *always* know when we're done reading a character.

PollEverywhere

Consider the following code

c	a	n	b	s
$E(c)$	0	10	110	100

What is the original text corresponding to the encoded text 1100100100?

pollev.com/comp526

Prefix Codes and Tries

Definition. A character encoding E is a **prefix code** if no codeword $E(c)$ is a *prefix* of another code

Example.

c	A	E	N	O	T	␣
$E(c)$	01	101	001	100	11	000

Prefix Codes and Tries

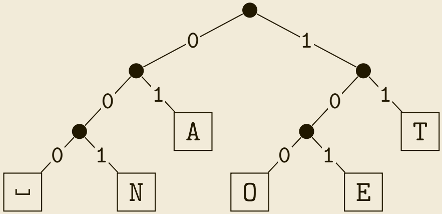
Definition. A character encoding E is a **prefix code** if no codeword $E(c)$ is a *prefix* of another code

Example.

c	A	E	N	O	T	␣
$E(c)$	01	101	001	100	11	000

Representation of prefix codes: the **trie** data structure!

- binary tree
- one leaf for each character
- edges labeled 0 or 1
- codewords = paths to leaves



Prefix Codes and Tries

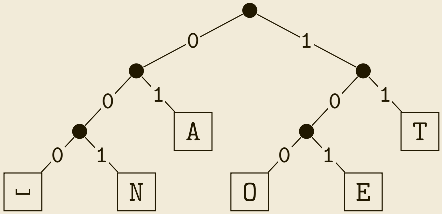
Definition. A character encoding E is a **prefix code** if no codeword $E(c)$ is a *prefix* of another code

Example.

c	A	E	N	O	T	␣
$E(c)$	01	101	001	100	11	000

Representation of prefix codes: the **trie** data structure!

- binary tree
- one leaf for each character
- edges labeled 0 or 1
- codewords = paths to leaves



Encoding. Use the *table:* `AN␣ANT`

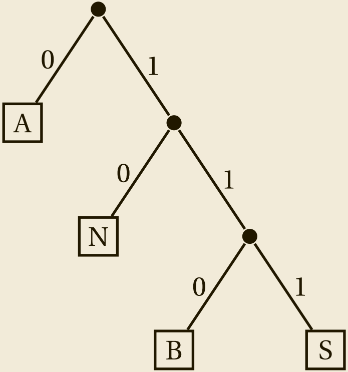
Decoding. Use the *trie:* `11100001010111`

Trie it Yourself

PollEverywhere Question

What is the result of using the trie on the right to decode the message:
`110010010011 1`

pollev.com/comp526



Fixed, Static, Adaptive

Note. In order to use a prefix code, we must also store the codewords!

- **fixed coding** uses the same code for all strings
  - e.g. ASCII, Unicode encodings (UTF-8)
- **static coding** uses the same codeword for each instance of a character in a text
  - codewords may different for different texts
  - must store/transmit the codewords as well as the encoded text!
- **adaptive coding** may change the codewords as the text is processed
  - codewords are stored implicitly within the coded message

Huffman Codes

Variable Length and Compression

Question. How can variable length encoding help with compression?

Variable Length and Compression

Question. How can variable length encoding help with compression?

Example. Consider the text `AAAAAAAAAAGGGH!`

- $\Sigma = \{$`A`,`G`,`H`,`!`$\}$
- Fixed length encoding:

c	`A`	`G`	`H`	`!`
$E(c)$	`00`	`01`	`10`	`11`

$\Longrightarrow$ Total encoded length = 30 (15 chars at 2 bits per char)

Variable Length and Compression

Question. How can variable length encoding help with compression?

Example. Consider the text `AAAAAAAAAAGGGH!`

- $\Sigma = \{$`A`, `G`, `H`, `!`$\}$
- Fixed length encoding:

c	`A`	`G`	`H`	`!`
$E(c)$	00	01	10	11

$\implies$ Total encoded length = 30 (15 chars at 2 bits per char)

- Exploiting frequency of `A` and `G`

c	`A`	`G`	`H`	`!`
$E(c)$	0	10	110	111

$\implies$ Total encoded length = 22

Variable Length and Compression

Question. How can variable length encoding help with compression?

Example. Consider the text `AAAAAAAAAAGGGH!`

- $\Sigma = \{$`A`,`G`,`H`,`!`$\}$
- Fixed length encoding:

c	`A`	`G`	`H`	`!`
$E(c)$	`00`	`01`	`10`	`11`

$\Longrightarrow$ Total encoded length = 30 (15 chars at 2 bits per char)

- Exploiting frequency of `A` and `G`

c	`A`	`G`	`H`	`!`
$E(c)$	`0`	`10`	`110`	`111`

$\Longrightarrow$ Total encoded length = 22

Question. How can we find the **best possible** prefix code for compression?

Exploiting Character Frequency

Generic Optimization Problem. Suppose we are given

- a string S over the alphabet Σ;
- weights $w(c) \geq 0$ for each $c \in \Sigma$.

Find the prefix code E for Σ that minimizes $\sum_c w(c)\,|E(c)|$

Exploiting Character Frequency

Generic Optimization Problem. Suppose we are given

- a string S over the alphabet Σ;
- weights $w(c) \geq 0$ for each $c \in \Sigma$.

Find the prefix code E for Σ that minimizes $\sum_c w(c)\,|E(c)|$

Example Weights. Take $w(c)$ to be the number of occurrences of c in S.

- note that $\sum_c w(c)\,|E(c)| = |E(S)|$
- so solving optimization problem gives the shortest possible (prefix code) encoding of S!

Exploiting Character Frequency

Generic Optimization Problem. Suppose we are given

- a string S over the alphabet Σ;
- weights $w(c) \geq 0$ for each $c \in \Sigma$.

Find the prefix code E for Σ that minimizes $\sum_c w(c)\,|E(c)|$

Example Weights. Take $w(c)$ to be the number of occurrences of c in S.

- note that $\sum_c w(c)\,|E(c)| = |E(S)|$
- so solving optimization problem gives the shortest possible (prefix code) encoding of S!

Question. Can we solve the optimization problem?

Exploiting Character Frequency

Generic Optimization Problem. Suppose we are given

- a string S over the alphabet Σ;
- weights $w(c) \geq 0$ for each $c \in \Sigma$.

Find the prefix code E for Σ that minimizes $\sum_c w(c)\,|E(c)|$

Example Weights. Take $w(c)$ to be the number of occurrences of c in S.

- note that $\sum_c w(c)\,|E(c)| = |E(S)|$
- so solving optimization problem gives the shortest possible (prefix code) encoding of S!

Question. Can we solve the optimization problem?

- I suppose we can with brute force: check all prefix codes
  - runs in exponential time in $|\Sigma|$

Exploiting Character Frequency

Generic Optimization Problem. Suppose we are given

- a string S over the alphabet Σ;
- weights $w(c) \geq 0$ for each $c \in \Sigma$.

Find the prefix code E for Σ that minimizes $\sum_c w(c)\,|E(c)|$

Example Weights. Take $w(c)$ to be the number of occurrences of c in S.

- note that $\sum_c w(c)\,|E(c)| = |E(S)|$
- so solving optimization problem gives the shortest possible (prefix code) encoding of S!

Question. Can we solve the optimization problem?

- I suppose we can with brute force: check all prefix codes
  - runs in exponential time in $|\Sigma|$
- Can we solve it *efficiently?*

Huffman Coding: Greed is Good

Idea. Build the character trie greedily from the leaves up.

- Prefix codes are binary trees with leaves labeled by Σ

Huffman Coding: Greed is Good

Idea. Build the character trie greedily from the leaves up.

- Prefix codes are binary trees with leaves labeled by Σ
- Maintain a collection A of active vertices
- Initially A is set of leaves, labeled with
    1. a character $c \in \Sigma$
    2. the weight $w(c)$

Huffman Coding: Greed is Good

Idea. Build the character trie greedily from the leaves up.

- Prefix codes are binary trees with leaves labeled by Σ
- Maintain a collection A of active vertices
- Initially A is set of leaves, labeled with
  1. a character $c \in \Sigma$
  2. the weight $w(c)$
- While $|A| > 1$:
  1. u and v are two lightest vertices
  2. add parent p to u and v
  3. set $w(p) = w(u) + w(v)$
  4. add p to A, remove u, v

Huffman Coding: Greed is Good

Idea. Build the character trie greedily from the leaves up.

- Prefix codes are binary trees with leaves labeled by Σ
- Maintain a collection A of active vertices
- Initially A is set of leaves, labeled with
  1. a character $c \in \Sigma$
  2. the weight $w(c)$
- While $|A| > 1$:
  1. u and v are two lightest vertices
  2. add parent p to u and v
  3. set $w(p) = w(u) + w(v)$
  4. add p to A, remove u, v

Example.

- $\Sigma = \{A, B, C, D, E\}$
- weights $= \{0.25, 0.15, 0.1, 0.1, 0.4\}$

LOSSLESS Example

Example. Find the Huffman encoding for the text `LOSSLESS`.

LOSSLESS Example

Example. Find the Huffman encoding for the text `LOSSLESS`.

Three Steps:

1. Compute frequency counts $w(c)$
2. Build Huffman tree
3. Write Huffman code from the tree

Huffman Analysis: Greed Works

Theorem

Given alphabet Σ and weight function $w : \Sigma \to \mathbf{R}_{\geq 0}$, the Huffman coding schemes gives the minimum weighted codeword length $\ell(E) = \sum_{c \in \Sigma} w(c) \cdot |E(c)|$ among all prefix codes.

Huffman Analysis: Greed Works

Theorem

Given alphabet Σ *and weight function* $w : \Sigma \to \mathbf{R}_{\geq 0}$*, the Huffman coding schemes gives the minimum weighted codeword length* $\ell(E) = \sum_{c \in \Sigma} w(c) \cdot |E(c)|$ *among all prefix codes.*

Proof sketch. Induction on $|\Sigma|$

- Let E^* be an optimal encoding/trie
- Claim: $\exists$ sibling leaves x, y at max depth
- Swap x and y for two min weight leaves, a, b
- Optimal code for $\Sigma' = \Sigma \setminus \{a, b\} \cup \{\boxed{\text{ab}}\}$ gives optimal code for Σ (verify this!)
- By inductive hypothesis, Huffman gives optimal code for Σ'
- So we get an optimal code for Σ $\square$

Huffman Computational Efficiency

Question. For an alphabet of size $m = |\Sigma|$ and weights w, how efficiently can we build the Huffman code?

- Maintain a collection A of active vertices
- Initially A is set of leaves, labeled with
  1. a character $c \in \Sigma$
  2. the weight $w(c)$
- While $|A| > 1$:
  1. u and v are two lightest vertices
  2. add parent p to u and v
  3. set $w(p) = w(u) + w(v)$
  4. add p to A, remove u, v
- Construct the codeword table

Tie Breaking Rules

So far we have two ambiguities in our Huffman trie description:

1. Which child is right/left child of the parent?
2. What do we do if weights are tied?

Tie Breaking Rules

So far we have two ambiguities in our Huffman trie description:

1. Which child is right/left child of the parent?
2. What do we do if weights are tied?

Conventions.

- Smaller weight child is on the left
- All ties broken by earliest character in alphabetical order
  - for internal vertices, the one containing the alphabetically first character as a descendant is on the left

Huffman and Entropy

A Thought Experiment

Suppose I have an alphabet $\Sigma = \{c_1, c_2, \ldots, c_n\}$ and I choose a character c_i *at random* to transmit

- each c_i is chosen with probability p_i.

A Thought Experiment

Suppose I have an alphabet $\Sigma = \{c_1, c_2, \ldots, c_n\}$ and I choose a character c_i *at random* to transmit

- each c_i is chosen with probability p_i.

Idea. Think of p_i as sub-intervals of $[0, 1]$.

- Outcome is a random point x in $[0, 1]$
- c_i corresponds to the interval containing x
- Use binary search to find the interval!

A Thought Experiment

Suppose I have an alphabet $\Sigma = \{c_1, c_2, \ldots, c_n\}$ and I choose a character c_i *at random* to transmit

- each c_i is chosen with probability p_i.

Idea. Think of p_i as sub-intervals of $[0, 1]$.

- Outcome is a random point x in $[0, 1]$
- c_i corresponds to the interval containing x
- Use binary search to find the interval!
- If the interval has width p_i need $\log(1/p_i)$ queries to determine interval
- The *expected* (average) number of queries is then

$$\mathscr{H}(p_1, p_2, \ldots, p_n) = \sum_{i=1}^{n} p_i \log\left(\frac{1}{p_i}\right)$$

- $\mathscr{H}$ is the **entropy** of the distribution over Σ

Properties of Entropy

Setup. We choose elements from $\Sigma = \{c_1, c_2, \ldots, c_n\}$ randomly, each c_i chosen with probability p_i.

One can show:

- Entropy $\mathcal{H}$ is a *lower bound* on the average number of bits needed to transmit a random character from Σ

Properties of Entropy

Setup. We choose elements from $\Sigma = \{c_1, c_2, \ldots, c_n\}$ randomly, each c_i chosen with probability p_i.

One can show:

- Entropy $\mathcal{H}$ is a *lower bound* on the average number of bits needed to transmit a random character from Σ
- If we use a Huffman encoding of Σ
  - weights $w(c_i) = p_i$
  - transmit the Huffman codeword $E(c_i)$

  Then the average length $\ell(E)$ of the transmitted word satisfies

$$\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$$

Properties of Entropy

Setup. We choose elements from $\Sigma = \{c_1, c_2, \ldots, c_n\}$ randomly, each c_i chosen with probability p_i.

One can show:

- Entropy $\mathcal{H}$ is a *lower bound* on the average number of bits needed to transmit a random character from Σ
- If we use a Huffman encoding of Σ
  - weights $w(c_i) = p_i$
  - transmit the Huffman codeword $E(c_i)$

  Then the average length $\ell(E)$ of the transmitted word satisfies

$$\mathcal{H} \le \ell(E) \le \mathcal{H} + 1$$

Conclusion. Huffman coding gives (nearly) the best possible *average* compression for *randomly* generated texts!

Emprical Entropy

Definitions. For a fixed string S over alphabet $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$, we define the **relative frequency** of character c_i in S to be

$$p_i = \frac{\#\text{ occurrances of } c_i \text{ in } S}{|S|}$$

The **empirical entropy** of S is then

$$\mathcal{H}_0(S) = \mathcal{H}(p_1, p_2, \ldots, p_\sigma).$$

Emprical Entropy

Definitions. For a fixed string S over alphabet $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$, we define the **relative frequency** of character c_i in S to be

$$p_i = \frac{\text{\# occurrances of } c_i \text{ in } S}{|S|}$$

The **empirical entropy** of S is then

$$\mathcal{H}_0(S) = \mathcal{H}(p_1, p_2, \ldots, p_\sigma).$$

The length of the Huffman encoded text $C = E(S)$ is

$$|C| = \sum_{i=1}^{\sigma} |S|_{a_i} |E(c_i)| = n \sum_{i=1}^{n} p_i |E(c_i)| = n\ell(E).$$

Applying the previous slide gives $\mathcal{H}_0(S)\, n \le |C| \le (\mathcal{H}_0(S) + 1)\, n$.

- Entropy and Huffman coding length are intimately connected

Next Time

More Compression!

- Limits of Compressibility
- Compressing Repetitive Texts

Scratch Notes