Module 10: Compression

CS 240 - Data Structures and Data Management

Sajed Haque Veronika Irvine Taylor Smith
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2017

Data Storage and Transmission

The problem: How to store and transmit data?

Source text	The original data, string of characters S from the *source alphabet* Σ_S
Coded text	The encoded data, string of characters C from the *coded alphabet* Σ_C
Encoding	An algorithm mapping source texts to coded texts
Decoding	An algorithm mapping coded texts back to their original source text

Note: Source "text" can be any sort of data (not always text!)

Usually the coded alphabet Σ_C is just binary: $\{0, 1\}$.

Judging Encoding Schemes

We can always measure efficiency of encoding/decoding algorithms.

What other goals might there be?

- Processing speed
- Reliability (e.g. error-correcting codes)
- Security (e.g. encryption)
- Size

Encoding schemes that try to minimize $|C|$, the size of the coded text, perform *data compression*. We will measure the *compression ratio*:

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

Types of Data Compression

Logical vs. Physical

- **Logical Compression** uses the meaning of the data and only applies to a certain domain (e.g. sound recordings)
- **Physical Compression** only knows the physical bits in the data, not the meaning behind them

Lossy vs. Lossless

- **Lossy Compression** achieves better compression ratios, but the decoding is approximate; the exact source text S is not recoverable
- **Lossless Compression** always decodes S exactly

For media files, lossy, logical compression is useful (e.g. JPEG, MPEG)

We will concentrate on physical, lossless compression algorithms. These techniques can safely be used for any application.

Character Encodings

Standard *character encodings* provide a matching from the source alphabet Σ_S (sometimes called a *charset*) to binary strings.

ASCII (American Standard Code for Information Interchange):

- Developed in 1963
- 7 bits to encode 128 possible characters:
  "control codes", spaces, letters, digits, punctuation
- Not well-suited for non-English text:
  ISO-8859 extends to 8 bits, handles most Western languages
- To decode ASCII, we look up each 7-bit pattern in a table.
- ASCII is called a *fixed-length code* because
  every key string in D has the same length (7 bits)

Other (earlier) codes: Morse code, Baudot code

Variable-Length Codes

Definition: Different key strings in D have different lengths

The UTF-8 encoding of Unicode provides a simple example:

- Encodes any Unicode character (more than 107,000 characters) using 1-4 bytes
- Every ASCII character is encoded in 1 byte with leading bit 0, followed by the 7 bits for ASCII
- Otherwise, the first byte starts with 1-4 1's indicating the total number of bytes, followed by a 0 and 3-5 bits. The remaining bytes each start with 10 followed by 6 bits.

Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
0000 0000–0000 007F	0xxxxxxx
0000 0080–0000 07FF	110xxxxx 10xxxxxx
0000 0800–0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000–0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Decoding

We need a a decoding algorithm mapping $\Sigma_C^* \to \Sigma_S^*$:

- A code must be *uniquely decodable*
- It is helpful if a code is **prefix-free** (why?)
- A prefix-free code has a dictionary $\Sigma_C* \to \Sigma_S$
- Dictionary:
  - Might be used and stored explicitly (e.g. as a trie), or only implicitly
  - Might be agreed in advance (*fixed coding*),
    stored alongside the message (*static coding*),
    or stored implicitly within the message (*adaptive coding*)

Character Frequency

Observation: Some letters in Σ occur more often than others.
So let's use shorter codes for more frequent characters.

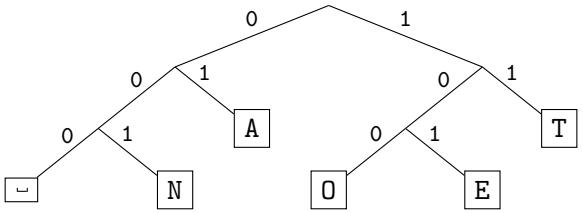
For example, the frequency of letters in typical English text is:

e	12.70%	d	4.25%	p	1.93%
t	9.06%	l	4.03%	b	1.49%
a	8.17%	c	2.78%	v	0.98%
o	7.51%	u	2.76%	k	0.77%
i	6.97%	m	2.41%	j	0.15%
n	6.75%	w	2.36%	x	0.15%
s	6.33%	f	2.23%	q	0.10%
h	6.09%	g	2.02%	z	0.07%
r	5.99%	y	1.97%		

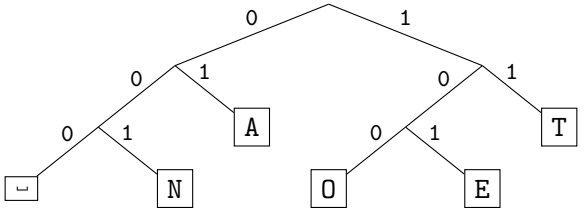
Huffman Coding

- Source alphabet is arbitrary (say Σ), coded alphabet is $\{0, 1\}$
- We build a binary trie to store the decoding dictionary D
- Each character of Σ is a leaf of the trie

Example: $\Sigma = \{\texttt{AENOT␣}\}$



Huffman Encoding/Decoding Example



- Encode `AN␣ANT`
- Decode `111000001010111`

Building the best trie

For a given source text S, how to determine the "best" trie which minimizes the length of C?

1. Determine the frequency of each character $c \in \Sigma$ in S
2. Make $|\Sigma|$ height-0 tries holding each character $c \in \Sigma$.
   Assign a "weight" to each trie: sum of frequencies of all letters in trie (initially, these are just the character frequencies)
3. Merge two tries with the least weights, new weight is their sum (corresponds to adding one bit to the encoding of each character)
4. Repeat Step 3 until there is only 1 trie left; this is D.

What data structure should we store the tries in to make this efficient?

Building trie example

Example text: `LOSSLESS`

Huffman Coding Summary

- Encoder must do lots of work:
  1. Build decoding trie (one pass through S, cost is $O(|S| + |\Sigma| \log |\Sigma|)$)
  2. Construct *encoding* dictionary from the trie mapping $\Sigma \to \{0,1\}^*$
  3. Encode $S \to C$ (second pass through S)
- Note: constructed trie is **not necessarily unique** (why?)
- Decoding trie must be transmitted along with the coded text C
- Decoding is faster; this is an *asymmetric* scheme.
- The constructed trie is an *optimal* one that will give the shortest C (we will not go through the proof)
- Huffman is the best we can do for encoding one character at a time.

Run-Length Encoding

- Variable-length code with a fixed decoding dictionary, but one which is not explicitly stored.
- Not a character-encoding (multiple characters represented by one dictionary-entry)
- The source alphabet and coded alphabet are both binary: $\{0,1\}$.

Observation: 0's and 1's in S may be repeated many times in a row (called a "run").

S is encoded as the first bit of S (either 0 or 1), followed by a sequence of integers indicating run lengths. (We don't have to encode the value of each bit since it will alternate.)

Question: How to encode a run length k in binary?

Prefix-free Integer Encoding

The encoding of run-length k must be *prefix-free*, because the decoder has to know when to stop reading k.

We will encode the binary length of k in **unary**, followed by the actual value of k in **binary**.

The binary length of k is $len(k) = \lfloor \log k \rfloor + 1$.
Since $k \geq 1$, we will encode $len(k) - 1$, which is at least 0.

The prefix-free encoding of the positive integer k is in two parts:

1. $\lfloor \log k \rfloor$ copies of 0, followed by
2. The binary representation of k

Examples: $1 \to 1$, $3 \to 011$, $5 \to 00101$, $23 \to 000010111$

RLE Example

$S =$ 11111110010000000000000000000011111111111

RLE Properties

- Compression ratio could be smaller than 1%
- Usually, we are not that lucky:
  - No compression until run-length $k \geq 6$
  - **Expansion** when run-length $k = 2$ or 4
- Method can be adapted to larger alphabet sizes
- Used in some image formats (e.g. TIFF)

Adaptive Dictionaries

ASCII, UTF-8, and RLE use *fixed* dictionaries.

In Huffman, the dictionary is not fixed, but it is *static*: the dictionary is the same for the entire encoding/decoding.

Properties of *adaptive encoding*:

- There is an initial dictionary D_0. Usually this is fixed.
- For $i \geq 0$, D_i is used to determine the i'th output character
- After writing the i'th character to output, both encoder and decoder update D_i to D_{i+1}

Note that both encoder and decoder must have the same information. Usually encoding and decoding algorithms will have the same cost.

Longer Patterns in Input

Huffman and RLE mostly take advantage of frequent or repeated *single characters*.

Observation: Certain *substrings* are much more frequent than others.

Examples:

- English text:
  Most frequent digraphs: `TH`, `ER`, `ON`, `AN`, `RE`, `HE`, `IN`, `ED`, `ND`, `HA`
  Most frequent trigraphs: `THE`, `AND`, `THA`, `ENT`, `ION`, `TIO`, `FOR`, `NDE`
- HTML: "`<a href`", "`<img src`", "`<br>`"
- Video: repeated background between frames, shifted sub-image

Lempel-Ziv

Lempel-Ziv is a family of *adaptive* compression algorithms.

Main Idea: Each character in the coded text C either refers to a single character in Σ_S, or a *substring* of S that both encoder and decoder have already seen.

Variants:

LZ77 Original version ("sliding window")
Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, ...
DEFLATE used in (`pk`)`zip`, `gzip`, `PNG`

LZ78 Second (slightly improved) version
Derivatives: LZW, LZMW, LZAP, LZY, ...
LZW used in `compress`, `GIF`
Patent issues!

LZW Overview

- Fixed-width encoding using k bits (e.g. $k = 12$).
  Store decoding dictionary with 2^k entries.
- First $|\Sigma_S|$ entries are for single characters,
  remaining entries involve multiple characters
- *Encoding:* after encoding a substring x of S, add xc to D where c is the character that follows x
- *Decoding:* after decoding a substring y of S, add xc to D, where x is previously encoded/decoded substring of S, and c is the first character of y
  Note: start adding to D after second substring of S is decoded

LZW Example

Input: `YO!␣YOU!␣YOUR␣YOYO!` Σ_S = ASCII character set (0–127)

$C =$

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	
129	
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	

LZW encoding pseudocode

```
LZW-encode(S)
1.   w ← NIL
2.   while there is input in S do
3.       K ← next symbol from S
4.       if wK exists in the dictionary
5.           w ← wK
6.       else
7.           output index(w)
8.           add wK to the dictionary
9.           w ← K
10.  output index(w)
```


LZW decoding

- Same idea: build dictionary while reading string.
- Example: 67 65 78 32 66 129 133 83

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)

LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- Generally: decoder is "one step behind" in creating dictionary
- So problem occurs if we want to use a code that we're about to build.
- But then we actually know what's going on!
  - Say we're seeing code-number k at the time when we're assigning k.
  - Encoder assigned $k \to s_{prev} + s[0]$
  - Decoder knows s_{prev} (decoded in previous step)
  - Decoder wants s, i.e., what k encodes, i.e., $s_{prev} + s[0]$
  - Therefore $s[0]$ = first character of s_{prev}
  - Therefore $s = s_{prev} + s_{prev}[0]$

LZW decoding pseudocode

```
LZW-decode(S)
1.   D ← dictionary that maps {0,...,127} to ASCII
2.   idx ← 128
3.   code ← first code from S
4.   s ← D(code); output s
5.   while there are more codes in S do
6.       s_prev ← s
7.       code ← next code of S
8.       if code == idx do
9.           s ← s_prev + s_prev[0]
10.      else
11.          s ← D(code)
12.      output s
13.      D.insert(idx, s_prev + s[0])
14.      idx ← idx + 1
```


LZW decoding example revisited

- Example: 67 65 78 32 66 129 133 83

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67,A
78	N	129	AN	65,N
32	␣	130	N␣	78,␣
66	B	131	␣B	32,B
129	AN	132	BA	66,A
133	ANA	133	ANA	129,A
83	S	134	ANAS	133,S

Compression summary

Huffman	**run-length encoding**	**Lempel-Ziv-Welch**
variable-length	variable-length	fixed-length
single-character	multi-character	multi-character
2-pass	1-pass	1-pass
60% compression on English text	bad on text	45% compression on English text
optimal 01-prefix-code	good on long runs (e.g., pictures)	good on English text
must send dictionary	can be worse than ASCII	can be worse than ASCII
rarely used directly (part of pkzip, JPEG, MP3)	rarely used directly (fax machines, old picture-formats)	frequently used (GIF, some variants of PDF)

Text transformations

For efficient compression, we need

- frequently repeating characters and/or
- frequently repeating substrings

Idea: Modify the text first so that these are likely.

Move-to-Front

Recall the MTF heuristic for self-organizing search:

- Dictionary is stored as an unsorted linked list
- After an element is accessed, move it to the front of the list.

How can we use this idea for text transformations?

Take advantage of *locality* in the data.
If we see a character now, we'll probably see it again soon.

Specifics: MTF is an *adaptive* compression algorithm.
If the source alphabet is Σ_S with size $|\Sigma_S| = m$,
then the coded alphabet will be $\Sigma_C = \{0, 1, \ldots, m-1\}$.

Move-to-Front Encoding/Decoding

MTF-encode(S)
1. $L \leftarrow$ linked list with Σ_S in some pre-agreed, fixed order
2. **while** S has more characters **do**
3. $\quad c \leftarrow$ next character of S
4. $\quad$ **output** index i such that $L[i] = c$
5. $\quad$ Move c to position $L[0]$

Decoding works in *exactly* the same way:

MTF-decode(C)
1. $L \leftarrow$ linked list with Σ_S in some pre-agreed, fixed order
2. **while** C has more characters **do**
3. $\quad i \leftarrow$ next integer from C
4. $\quad$ **output** $L[i]$
5. $\quad$ Move $L[i]$ to position $L[0]$

MTF Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$ `INEFFICIENCIES`

$C =$

- What does a run in S encode to in C?
- What does a run in C mean about the source S?

MTF Compression Ratio

So far, MTF does not provide any compression *on its own* (why?)

We need to encode the integer sequence.

Two possible approaches:

- Prefix-free integer encoding (like in RLE)
- Huffman coding

Burrows-Wheeler Transform

The Burrows-Wheeler Transform is a sophisticated compression technique

- Transforms source text to a coded text with the same letters, just in a different order
- *The coded text will be more easily compressible with MTF*
- Compression algorithm does not make just a few "passes" over S. BWT is a *block* compression method.
- (As we will see) decoding is more efficient than encoding, so BWT is an *asymmetric* scheme.

BWT (followed by MTF, RLE, and Huffman) is the algorithm used by the `bzip2` program.

It achieves the best compression of any algorithm we have seen (at least on English text).

BWT Encoding

A *cyclic shift* of a string X of length n is the concatenation of $X[i+1..n-1]$ and $X[0..i]$, for $0 \leq i < n$.

For Burrows-Wheeler, we assume the source text S ends with a special *end-of-word character* \$ that occurs nowhere else in S.

The Burrows-Wheeler Transform proceeds in three steps:

1. Place all *cyclic shifts* of S in a list L
2. Sort the strings in L lexicographically
3. C is the list of trailing characters of each string in L

BWT Example

$S =$ `alf␣eats␣alfalfa$`

1. Write all cyclic shifts
2. Sort cyclic shifts
3. Extract last characters from sorted shifts

C =

BWT Decoding

Idea: Given C, We can generate the *first column* of the array by sorting. This tells us which character comes after each character in S.

Decoding Algorithm:
View the coded text C as an array of characters.

1. Make array of A of tuples $(C[i], i)$
2. Sort A by the characters, record integers in array N (Note: $C[N[i]]$ follows $C[i]$ in S, for all $0 \leq i < n$)
3. Set j to index of \$ in C and S to empty string
4. Set $j \leftarrow N[j]$ and append $C[j]$ to S
5. Repeat Step 4 until $C[j] = \$$

BWT Decoding Example

$C =$ `ard$rcaaaabb`
$S =$

A		$sort(A)$		N
a, 0		$, 3		3
r, 1		a, 0		0
d, 2		a, 6		6
$, 3		a, 7		7
r, 4		a, 8		8
c, 5	$\longrightarrow$	a, 9	$\longrightarrow$	9
a, 6		b, 10		10
a, 7		b, 11		11
a, 8		c, 5		5
a, 9		d, 2		2
b, 10		r, 1		1
b, 11		r, 4		4

BWT Overview

Encoding cost: $O(n^2)$ (using radix sort)

- Sorting cyclic shifts is equivalent to sorting suffixes
- This can be done by traversing suffix trie
- Possible in $O(n)$ time

Decoding cost: $O(n)$ (faster than encoding)

Encoding and decoding both use $O(n)$ space.

Tends to be slower than other methods but (combined with MTF and Huffman) give better compression.