Computer Science Basics
Compression of Files

Fall Term 2025/2026

Emmanuel Benoist | BFH-TI

Encoding and Compression of Files

- **Last Week**
- **Lossless compressions**
  RLE Run-Length Encoding
  Huffman
  LZW - Lempel-Ziv-Welch
- **Conclusion**

Last Week

Character Encoding

UTF-8 is the standard now

- Used in Web sites

Problem for Databases

- UTF-8 characters do not have the same length
- Not possible to be stored in a DB
- MySQL / MariaDB : Characters are encoded on 3 bytes.
- MS SQL Server found another solution.

Legacy systems use many different encodings

- Unix / Mac / Windows
- US / latin-1 / latin-???
- International systems : UTF-16BE, UTF-16LE, UTF-32...

Job of the Medical Informatics specialist

- Let all the systems work together!

Lossless compressions

Compression

Large data

- Texts
- Web Pages
- Programs
- Documents
- Images
- Sounds
- Videos

Lossless compression

- Data are compressed, without losing any information
- Texts, web pages, programs (source or executables), documents
- zip, gZip, …

Lossy compression

- For images or Videos
- .jpeg, different videos codecs, different audio codecs.

RLE - Run-Length Encoding

How to compress

`AAAAAAAAAAAAAAAA`

Solution

`16A` **Examples**

- `AAABBBCCC` becomes `3A3B3C`

Less good

- `Another example` becomes `1A1n1o1t1h1e1r1SP1e1x1a1m1p1l1e` (2 x larger)

RLE for Black and White images

If we have an image with only Black and White pixels[1]

- Black pixels are denoted B
- White pixels are denoted W

The image

- `WWWWWWWWWWWWBWWWWWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWW`

Becomes

- `12W1B14W3B23W1B11W`

[1] Source: `https://fr.wikipedia.org/wiki/Run-length_encoding`

Applications

Images

- BMP format uses compression for 1bit, 4 and 8 bits per pixel (Black and White, 16 colors, 256 colors).
- PCX uses 8 and 24 bits per pixel (24 bits mean 3 channels with one byte per pixel for RGB colors).

Fax

- Only the lengths are sent, saving more place
- Every line must start with the same color.

Huffman Code

Invented in 1952 by David Albert Huffman

Idea

- Same as Morse
- Use less bit for characters that occur more often
- Example in Morse e is coded ‘‘.’’

First step: read the text and build a tree

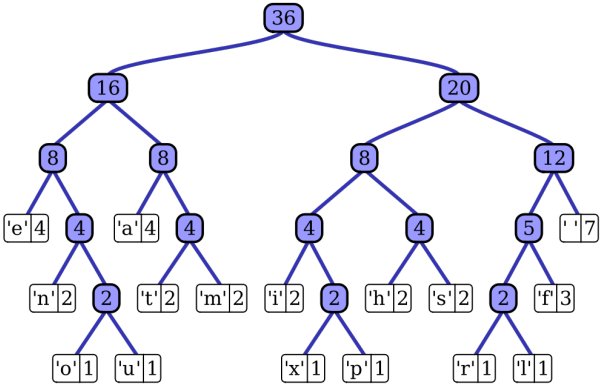
- The elements occurring more often are high in the tree (e, s, t in english),
- Elements occurring rarely are low in the tree (k,y,z in english).

Second step: use the tree to encode characters

- We use the path between root and the element to describe the node
- The higher a letter is, the shorter its coding.

Huffman code tree

Tree for the sentence “This is an example of a huffman tree” [2]



[2] Source `https://fr.wikipedia.org/wiki/Codage_de_Huffman`

Which tree?

Always the same tree (depending on the type of file)

- For french text
- For english text
- For german text
- For a Java Program

A tree specific for one document

- Text is read once, tree is generated
- Code is the smallest possible
- But tree needs to be sent

Adaptative method

- One default tree is modified dynamically by reading the text
- Tree remains small
- Larger computation (need to modify the tree during decoding)

Encoding/Decoding

Each time a byte is read

- Search in the tree

One writes the bits corresponding to the path in the tree

- Starting at the root
- `0` means left
- `1` means right

Example

- `emmanuel` is coded `000 0111 0111 010 0010 00111 000 11001`
- It uses 31 bits, whereas 8 x 7 bit are needed in Ascii

LZW - Lempel-Ziv-Welch

Algorithm [3]

- Based on LZ78 written by Abraham Lempel et Jacob Ziv
- Enhanced by Terry Welch

Compression based on a dictionary

- The dictionary is created and maintained during reading
- We replace common words by their place in the dictionary
- The dictionary is initialized with all the bytes (every word has one character).

If some words are repeated, compression is efficient

[3] Source: `http://igm.univ-mlv.fr/~dr/XPOSE2013/La_compression_de_donnees/lzw.html`

Principle

You reed the file and construct the dictionary

- Need only one pass

Dictionary contains all the one byte words per default

- For a text all ASCII (or better latin1) characters.

If a word exists in the dictionary, it is replaced by its rank in the dictionary If a new word is read, then it is added in the dictionary for later use

Example: Encode a String

Encode `TOBEORNOTTOBEORTOBEORNOT`[4]

Dictionary initially contains 256 words of on byte (all the possible chars).

w	c	wc	Output	dictionary
	T	T		
T	O	TO	T	TO = <256>
O	B	OB	O	OB = <257>
B	E	BE	B	BE = <258>
E	O	EO	E	EO = <259>
O	R	OR	O	OR = <260>
R	N	RN	R	RN = <261>
N	O	NO	N	NO = <262>
O	T	OT	O	OT = <263>
T	T	TT	T	TT = <264>
T	O	TO		
TO	B	TOB	<256>	TOB = <265>
B	E	BE		
BE	O	BEO	<258>	BEO = <266>
O	R	OR		
OR	T	ORT	<260>	ORT = <267>
T	O	TO		
TO	B	TOB		
TOB	E	TOBE	<265>	TOBE = <268>
E	O	EO		
EO	R	EOR	<259>	EOR = <269>
R	N	RN		
RN	O	RNO	<261>	RNO = <270>
O	T	OT		
OT			<263>	

[4] `https://fr.wikipedia.org/wiki/Lempel-Ziv-Welch`

Example: Encoded output

The output of the algorithm is:

- `TOBEORNOT<256><258><260><265><259><261><263>`

Place needed

- Normally (ASCII): 24 bytes, i.e. 192 bits
- Each number is encoded on 9 bit (largest number is 263 (< 512)
- 16*9 bit = 144 bits

Example: Decoding the text

- **Input:** `TOBEORNOT<256><258><260><265><259><261><263>`

c	w	ln	w+in[o]	Output	dictionary
T	T			T	
O	T	O	TO	O	TO = <256>
B	O	B	OB	B	OB = <257>
E	B	E	BE	E	BE = <258>
O	E	O	EO	O	EO = <259>
R	O	R	OR	R	OR = <260>
N	R	N	RN	N	RN = <261>
O	N	O	NO	O	NO = <262>
T	O	T	OT	T	OT = <263>
<256>	T	TO	TT	TO	TT = <264>
<258>	TO	BE	TOB	BE	TOB = <265>
<260>	BE	OR	BEO	OR	BEO = <266>
<265>	OR	TOB	ORT	TOB	ORT = <267>
<259>	TOB	EO	TOBE	EO	TOBE = <268>
<261>	EO	RN	EOR	RN	EOR = <269>
<263>	RN	OT	RNO	OT	RNO = <270>

Use of LZW algorithm

For images compression

- Images in Gif, TIFF,
- Audio files MOD

Characters have 12 bits (not 8)

- A pixel is 3 bytes (RGB) i.e. 24 bit
- 2characters represent one pixel

Very good at recognizing patterns

- But now, images have millions of colors and do not have that much patterns.

Maybe we could change a little bit the image to help the compression

- Lossy compression : Next week

Conclusion

Conclusion

Compression is central in a lot of Med Inf topics

- Storage of large documents
- Transfer of large documents

Algorithms

- RLE is efficient if their are a lot of repetitions (Black and White images for instance).
- Huffman generates a tree. Is efficient if the type is known or if the tree remains small regarding the size of the document.
- LZW is efficient even if nothing is none.

ZIP files

- For lossless compression of files
- Uses the algorithm Deflate based on LZ77 and Huffman
- First compress the duplicate series of bytes (Lempel-Ziv)
- Then replaces commonly used symbols (Huffman).
- Very efficient with text files.

Lossless vs lossy compression

- Sometime, we do not need all of the information, a part of it is sufficient

References

Web pages

- https://fr.wikipedia.org/wiki/Codage_de_Huffman
- https://fr.wikipedia.org/wiki/Lempel-Ziv-Welch
- http://www.journaldunet.com/developpeur/tutoriel/theo/041014-algo-compression-sans-perte.shtml
- https://www.numerama.com/tech/299921-comment-fonctionne-la-compression-de-donnees.html
- http://igm.univ-mlv.fr/~dr/XPOSE2013/La_compression_de_donnees/lzw.html

Exercises

1. **Encode using RLE**
   - ‘‘Hallo Mitenand’’
   - ‘‘AAAAAABBBBBAAAACCCCCWWWWW’’
2. **Encode using LZW**
   - ‘‘Bonjour a vous tous’’
   - ‘‘GTACCTAGGTAGTAAGTATGTAC’’