Lecture 09

Image Compression

Haitham A. El-Ghareeb

June 10, 2019

Faculty of Computers and Information Sciences
Mansoura University
Egypt
`helghareeb@mans.edu.eg`

Contacts

- https://www.haitham.ws
- https://youtube.com/user/helghareeb
- https://www.github.com/helghareeb
- http://eg.linkedin.com/in/helghareeb
- helghareeb@mans.edu.eg

Introduction

Introduction

- The advantages of using data compression are pretty straightforward
- It lets you store more stuff in the same space
- it lets you transfer that stuff in less time, or with less bandwidth
- for pure data compression algorithms, the compressed file contains exactly the same amount of information as the original one

Data vs. Information

Data vs. Information

- There is a distinction between information and data
- The same e-mail twice

Digital Data Compression

- The fundamental idea is to take a given representation of information (a chunk of binary data)
- and replace it with a different representation (another chunk of binary data) that takes up less space
- (space here being measured in binary digits, better known as bits),
- and from which the original information can later be recovered.

Lossless vs. Lossy

- **Lossless** If the recovered information is guaranteed to be exactly identical to the original
- **Lossy** If the recovered information is <u>not</u> guaranteed to be exactly identical

Run-Length Encoding (RLE)

RLE

- Let's say we have a text file consisting of the following characters:

```
AAAAAAAAAABBCDEEFFFFFFFFFFFFFGGGGGGGGGGGGGGGGG
```

RLE Mechanism of Action

- RLE replaces ”runs” (that is, sequences of identical characters) with a single character, followed by the ”length of the run” (the number of characters in that sequence),
- or vice-versa (first the length and then the character)
- The order isn’t important as long as it’s always the same).

RLE Effect

- If we apply this algorithm to the text file above, we'll get something like this:

```
10A 2B 1C 1D 2E 13F 17G
```

- We have just gone from 46 characters to 17,
- and that's using a plain text representation for the lengths
- Using spaces would not be necessary

How many Bytes?

- 1 Byte for character
- Single byte can represent run lengths up to 255 characters long / 256 if 0 is used
- Required Bytes = 14 Bytes (7 lengths and 7 Characters, each pair = 2 Bytes)

Let's Go
Home

Run-Length Encoding (RLE)

Problem 01

Problem 01

- not all sequences of characters became shorter using this new representation
- Example
  - "BB" became "2B"
  - takes up the same two bytes
  - "C" actually doubled in size (from one to two bytes), becoming "1C"

Real Example

- Consider the following statement

```
Mary had a little lamb.
```

- It becomes

```
1M 1a 1r 1y 1  1h 1a 1d 1  1a 1  1L 1i 2t 1l 1e 1
   1L 1a 1m 1b 1.
```

Note

- In other words, the only part that doesn't get bigger is the "tt" in "little",
- and even that just manages to stay the same size (2 bytes).
- So we've just gone from 23 bytes (23 characters) to 44 (remember, spaces in the original sentence must be encoded, too,
- otherwise they won't be there when we decompress it).
- Not exactly brilliant, as far as "compression" goes.

Total Failure?

So RLE is a Total Failure!

Invoice

Item X

. . . .

$100.00

Notes

- there are 60 dots between the item name and its price
- "Item X" would double in size (from 6 to 12 bytes)
- "$100.00" would also grow
- the sequence of dots would go from 60 to 2 bytes
- the complete line would go from 73 bytes to 24
- That's a reduction to about 1/3rd of the original size

Challenge

- But if the same file contained a couple of paragraphs of ”normal” text, with few or no repetitions, all our gains would be lost.

Solution

- we’d like to have a way to compress only some parts of the file, while leaving others uncompressed.
- Can we?

Yes!

- Define a special ”marker” that indicates the beginning or the end of each block of compressed text
- For example, let’s say we use an asterisk as our marker, and determine that compression is off by default and will only be turned on when there are three or more identical characters in a row.

Enhanced RLE

```
Item X*[60].*\$100.00
```

- That's just 17 bytes
- (the square brackets indicate that the number between them is stored as a value, and not as a sequence of decimal digits;
- the value "60" can be represented by a single byte, in a computer file).

Enhanced RLE for Mary

```
Mary had a little lamb.
```

- No gain, but no loss, either

Let's Go
Home

Run-Length Encoding (RLE)

Problem 02

Problem 02

- What if the original file contained an asterisk?
- Passing * directly confuses the decompressor

Solution 01

```
Figaro was the city's factotum*.
```

```
Figaro was the city's factotum*[1]**.
```

- Increased the size
- as Encoder and Decoder both know the rules, it works!

Solution 02

- Use the rarest character in written English as the marker
- Use Null
- **Not the best Options!**

Back to the *

- Input

```
***
```

- If toggle is on

```
[3]*
```

- If toggle is off

```
*[3]*
```

Example 02

- Input

```
AAA**DDDD
```

- Output

```
*[3]A[2]*[4]D
```

Example 03

- Input

```
ABC**DEFG
```

- Output

```
ABC*[2]**DEFG
```

* inside

- If Previous Byte was a Run Length, * is a character
- If Previous Byte was a character, * is a toggle

Let's Go Home

Run-Length Encoding (RLE)

Problem 03

Problem 03

- Input

```
AAAHH!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!???
```

- Output

```
*[3]A[2]H[42]![3]?
```

The Cath

Where is the Catch?

ASCII

- the byte [42] happens to be the ASCII code for the asterisk
- the sequence above would be seen by the decoder as

```
∗[3]A[2]H∗![3]?
```

- the decoder would interpret the counter [42] as a compression toggle

NULL as a Toggle

- Byte with the value zero (often called a NULL) is usually an excellent choice as a default marker
- since we would probably never want to use zero as a run length anyway.
- However, if the source file we are trying to process contains a lot of null bytes, we might end up wasting a significant amount of space encoding those,
- so scanning the source for the rarest character is still the preferred solution.

Let's Go Home

One Last Optimization

- Sequences of exactly two identical characters
- those sequences always take up two bytes
- AA or [2]A
- the decision to compress them or not should (ideally) be based on whether they appear immediately after compressible data or not

Example 01

- Input

```
ABCDZZ
```

- Output

```
ABCDZZ
```

- the "ZZ" sequence would be treated as uncompressible

Example 02

- Input

```
AAAAZZ
```

- Output

```
*[4]A[2]Z
```

- the "ZZ" sequence would be treated as compressible

Challenge

Can you write an implementation of RLE?

Image Compression

Image Compression

- There is one type of data where very long sequences of identical values are common.
- Images, specifically about high-contrast images with few colors,
- such as schematic drawings, logos, text scans (including most faxes) and cartoons.
- It's also common, in some Computer Graphics animations, to use a plain background (to be replaced later),
- meaning that hundreds or even thousands of sequential pixels have exactly the same color.

RLE Advantages and Usage

- RLE has the advantage of being very simple
- therefore very fast to execute
- TARGA (.TGA extension) image file format can use RLE compression
- BMP, PCX, PackBits (a TIFF sub-format) and ILB
- Fax machines also use RLE

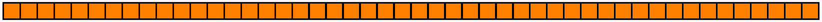
Image Compression

RGB

RGB

- RGB - Three Channels
- RGB Alpha - Fourth Channel - Opacity
- The alpha channel is usually the easiest one to compress, since most pixels tend to be 100% opaque or 100% transparent

RLE for Image Compression



- Image Compression Ratio of this image using RLE?

Answer

- Zero %
- Following sequence will repeat itself thousands of times (307200 times, for a 640x480 image)
- Image is represented as follows

```
[255][128][0][255][128][0]  ...etc.
```

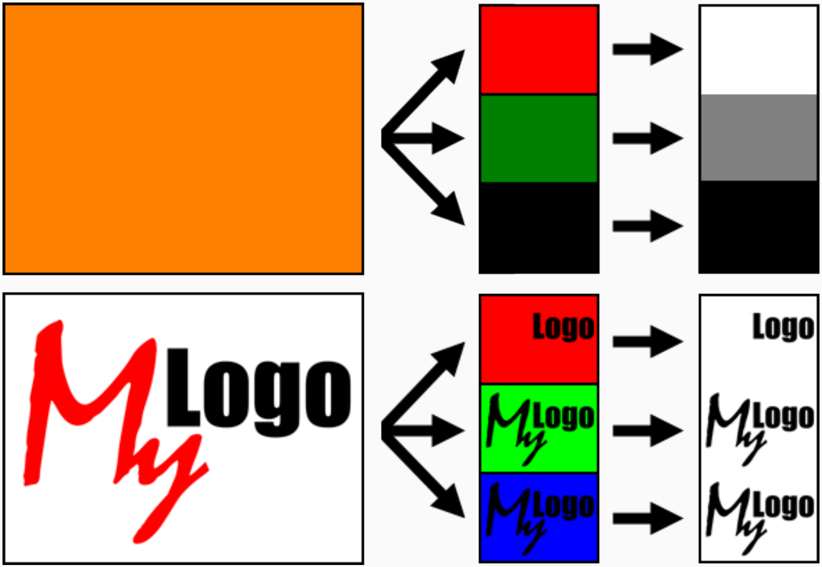
Solution 01

- Use Three Bytes instead of one
- Rewrite the algorithm to use sequences of 24 bits (3 bytes) instead of 8 bits (1 byte)
- Custom solution for a specific image, not a general one
- There is a simpler one

Solution 02

- Sorting the pixel color data by pixel by channel
- instead of having a file with 307200 repetitions of [255] [128] [000],
- we have a file with 307200 repetitions of [255], followed by 307200 repetitions of [128], followed by 307200 repetitions of [0].
- The full pixel data goes from 900kB to just over 7kB. That's a reduction of more than 99%!

Sorting the Channels



Let's Go Home

Challenge

- TGA uses this Trick
- We need to update the decoder
- Suitable for plain images
- Will never achieve high levels of compression on photographic images, or any other kind of image where there are no adjacent pixels with the same color

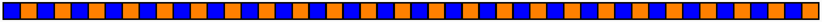
Lempel-Ziv Compressor Family

Lempel-Ziv

Suitable of files where there are

- unknown length of the patterns
- multiple patterns with different lengths

Example



```
[0][0][255] [255][128][0] [0][0][255] [255][128][0]
    ... etc.
```

- This sequence might seem slightly compressible,
- since there is a sequence of three repeated values followed by a sequence of two repeated values that appears again and again in our file.

Let's Go Home

Lempel-Ziv Compressor Family

Problem 04

Problem 04

- Having to constantly switch between compressed and uncompressed mode would actually make the file larger:

```
[0] [0][0][255] [255] (5 bytes)
```

- would be converted into

```
∗[3][0][2][255]∗ (6 bytes, including the markers)
```

Enhanced RLE

- Applying our "RGB optimisation" (sorting the data by channel) wouldn't help, either. We'd get this:

```
R:  [0][255][0][255][0][255]  ...etc.
G:  [0][128][0][128][0][128]  ...etc.
B:  [255][0][255][0][255][0]  ...etc.
```

- Still no sequences of identical values
- RLE wouldn't be able to compress it.

Solution

- if the compressor could look for data patterns, regardless of their length, and replace the repetitions of those patterns with some sort of ”counter”

Example 01

```
All these effects are available when working with
    high dynamic range (HDR) images.
```

```
To avoid clipping the highlights, make sure HDR
    mode is enabled.
```

Lempel-Ziv

- Abraham Lempel is a computer scientist
- Jacob Ziv is an electrical engineer.
- Together, in 1977 they published an algorithm, called LZ77, based on some of the principles described above.
- A slightly different version was published in 1978, under the name LZ78.
- And perhaps its most famous variant was published in 1984 by Terry Welch, under the name LZW.

How they Work?

- ”dictionary-based compressors”
- They work by replacing redundant (i.e., repeated) source data with references to its previous appearance (LZ77)
- or by explicit references to a ”dictionary” compiled from all the data in the source file (LZ78).

Example 02

```
The Flying Spaghetti Monster is real. I believe in
    the Flying Spaghetti Monster.
```

- Pair would be:
    - Length = 27
    - Distance = 54
    - Marker = *

```
The Flying Spaghetti Monster is real. I believe in
    t*[27][54]*.
```

- Decompressor will step back 54 characters and copy a 27-character sequence into the new position

Sliding Window Algorithm

- This technique (of looking back and copying sequences using a length-distance pair) is called a ”sliding window” algorithm.
- It’s a special type of dictionary algorithm, where the dictionary is built along the way, and older sequences are forgotten (as they fall out of the ”window”).

Lempel-Ziv Compressor Family

Problem 05

Problem 05

- If we chose to use a single byte to store the distance (of the length-distance pair), then we'd be limited to going back 256 characters, and there wouldn't be any point in keeping older sequences.
- If we chose to use a 2-byte value (known as a "short integer", or "int16") instead, then we could go back up to 65536 characters, and so on.
- The bigger the size of the window, the more efficient the compression will be, but it will require more RAM (system memory) to run.

Pure Dictionary based Encoder

- The sequences are stored at the beginning of the compressed file,
- and remembered for all the duration of the compression and decompression.
- That allows the compressor to achieve better results with the same amount of RAM, but requires extra processing
- (to create the dictionary, the compressor needs to analyze all the source data before it starts to actually compress it).

Lempel-Ziv Compressor Family

LZ77

LZ77

```
[255][128][0] * [921597][3] *
```

- Length = 921597
- Distance = 3

Let's Go Home

Lempel-Ziv Compressor Family

Problem 06

Problem 06 - a

- We can't store the number 921597 as a single byte (the maximum would be 255 - assuming we don't reassign zero),
- and we can't even store it as an short integer (the maximum would be 65535),
- so let's assume we use a 4-byte value (known as an "integer", or "int32"), that can store values from 0 to 4294967295.

Problem 06 - b

This means we have

- Three source data bytes,
- followed by a 1-byte marker,
- followed by a 1-byte length,
- followed by a 4-byte distance,
- followed by another 1-byte marker (which isn't really necessary, since there's no data after it, but let's include it anyway.
- This adds up to 10 bytes.

Comments

- Although this might seem a lot better than the result we got from our RLE compressor,
- the comparison is unfair, because we limited the RLE compressor to using a single byte to store the length for each sequence, and we are allowing the LZ compressor to use 4-byte integers.
- If we had allowed our RLE encoder to use 4-byte values, it would have been able to compress the raw image data into 15 bytes; not as good as the LZ algorithm, but pretty close.

Lempel-Ziv Compressor Family

LZ78

LZ78

```
Dictionary: [X][255][128][0]
Reference list: [921597][X]
```

- Where X is a code generated by the encoder
- It's harder to measure the exact size here

Summary

Summary

- Data vs. Information
- Data vs. Image Compression
- RLE
- Dictionary based Algorithms (Sliding Window)
- LZ Family
- Different Problems and Solutions

Let's Go
Home

https://www.haitham.ws