0.1 PNG Encoding and Decoding

- *Algorithm:* Filtering (algo. 1), Huffman coding (algo. 3), LZ77 (algo. 4)
- *Input:* Origin raw image as pixelwise color data.
- *Complexity:* Not the main topic.
- *Data structure compatibility:* Array, Huffman tree
- *Common applications:* Lossless image compression.

Problem. PNG Encoding and Decoding

The portable network graphic (PNG) format is a bitmap image file format which is portable and supports lossless compressed [1]. It is developed to replace Graphics Interchange Format (GIF), but posses more features that GIF does not have [1]. To increase space efficiency, it do filtering on th original bitmap data and perform DEFLATE algorithm for compression. This allows PNGs to carry image information with much smaller size, compared to formats like GIF.

Description

The most instinctive way to store an image file is to direct save the pixel information as a large 2 or 3-dimensional array (for gray-scale and RGB images). However, this method occupies large space and is not suitable for transmission, especially through the internet. PNG brings an encoding method to significantly compress the original information. It first filters on the raster graphics and then apply an in-the-box compression algorithm called DEFALTE. We will first talk about these two aspects.

Filtering

Before introducing the principles of filtering, we will discuss instinct of coming up with filtering. For a series of data, if they are linearly correlated, i.e. the difference between consecutive elements are low, then those differences would have high probability to be lower than the original data and many of them are duplicated. This provides convenience for further compressions. For example, a transformation on an integer series is shown below:

$$a = [2, 3, 4, 6, 7, 8, 6, 5, 4, 3, 1] \Rightarrow b = [2, 1, 1, 2, 1, 1, -2, -1, -1, -2]$$

We may see that an evenly distributed array is transformed into duplicated integers like ± 1 or ± 2. This make the data more compressible. In this simple case, we calculates the difference of neighbouring elements, and such method is called Delta encoding. PNG make use of adjusted Delta encodings, which is named "filtering", where it calculates the differences of the current pixel with a predictor based on the pixel to the left, above and above-left [6]. The brief process of filtering is described in Algorithm 1. There are 5 type of filtering in PNG, which are shown in Table 1. Denote the current pixel as X, and the left, top and top-left pixel as L, T and TL.

The type 4, Paeth predictor is due to Alan W. Paeth [7], which first calculate a simple linear composition of the three pixels, then choose the closest pixel to it from the 3 original ones as the predictor. A simple psuedocode illustrates it in Algorithm 2 [9].

Despite the way of demonstration here, true filtering process the data as byte instead of pixels, regardless of the bit depth or color type. By line scanning on the image, byte streams are created, where filtering algorithm will be working on.

Algorithm 1: Filtering

Input: Image lines L_1, L_2 with length n, where L_1 is right above L_2
Output: The filtered line of L_2

$L_2 = []$
$L_2'[1] = L_2[1]$
for $i \leftarrow 2$ **to** n **do**
  $L_2'[i] = L_2 - Predictor(L_2[i-1], L_1[i], L_1[i-1])$
end for
return L_2'

Type	Description	Expression
0	No filtering	X
1	With left	$X - L$
2	With top	$X - T$
3	With Avg of left and top	$X - \lfloor (L + T)/2 \rfloor$
4	Paeth predictor	$X - P(L, T, TL)$

Table 1: Types of PNG filtering

Algorithm 2: PaethPredictor

Function `PaethPredictor`(X, L, T, TL)**:**
  $P = L + T - TL$;
  $pl = |P - L|$; $pt = |P - T|$; $ptl = |P - TL|$;
  if pl *is minimum* **then**
    return L
  else if pt *is minimum* **then**
    return T
  else
    return TL
  end if
end

Each filter has its best performance cases, and the encoder programs would automatically choose the best one given the image rows. Such a row-by-row choice improves the compression. This heuristic method of improving is designed by Lee Daniel Crocker, one of the member of PNG working group, who performed empirical tests on many images during the creation of PNG format [2].

DEFLATE compression

DEFLATE is a lossless compression file format, designed by Phil Katz. Primary for personal use of a archiving tool, but was later specified as standard in RFC 1996 [3]. Technically, it's a combination of Huffman coding and LZSS compression algorithm. The former increase the space efficiency by attributing shorter coding for more frequently appeared characters. The latter achieves compression by reusing duplicated parts in the data. We are discussing these 2 algorithms in the following parts.

Huffman code is a type of prefix code used for lossless compression, while Huffman coding describes the algorithm of finding such a code, designed by David A. Huffman [5]. Prefix code is defined on a system such that no element is the prefix of another. For example, the set $\{1, 2, 12\}$ doesn't satisfy the prefix property, as 1 is the prefix of 12. This means that a Huffman code can be generated by a binary trees, called Huffman tree, and the Huffman coding contains the way to generate a Huffman tree and develop a coding system from it. It is shown in the Algorithm 3. It takes in the frequency statistics and returns the binary codes for each character.

Algorithm 3: Huffman Coding

Input : Character frequency statistics as a dictionary F
Output: Codes for each character as a dictionary C

```
Function HuffmanTree(F):
    Q ← a min-queue
    foreach character ch do
        create new node n
        n.char = ch; n.value = F[ch]
        Q.Insert(n, n.value)
    end foreach
    while Q.size > 1 do
        create a new node n
        n.left ← Q.ExtractMin()
        n.right ← Q.ExtractMin()
        n.value ← n.left.value + n.right.value
        Q.Insert(n, n.value)
    end while
    root = Q.ExtractMin()
    return root
end
Function GenerateCode(node):
    if node == root then return ""                    /* can be improved with dynamic programming */;
    else if n == n.parent.left then return GenerateCode(node.parent) + "0";
    else if n == n.parent.right then return GenerateCode(node.parent) + "1";
end
tree ← HuffmanTree(F)
foreach unique character ch do
    n ← the leaf node such n.ch == ch C[ch] = GenerateCode(n)
end foreach
return
```

Here offer an example of encoding the characters with frequency of

$$[a : 8,\ b : 6,\ c : 1,\ d : 3,\ e : 9]$$

which would gives a Huffman tree in Figure 1. The left path corresponds to 0 and right to 1, so we can obtain the coding table shown in Table 2. Note that no element is the prefix of another. Since if code c_1 is the prefix of code c_2, the node of c_1 should be an ancestor of that of c_2, which cannot be a leaf itself, so no character would be encoded as c_2. This guarantee the possibility of decoding.

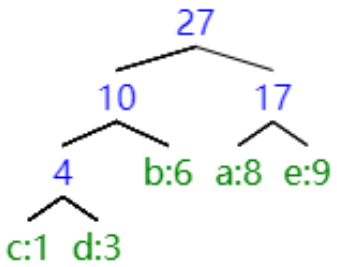

Figure 1: Example Huffman Tree

We need 3 bits originally to encode each of the 5 characters, which would take up $3 \times (8+6+1+3+9) = 81$ bits. However, with the Huffman code offerred in Table 2, the new size is $2 \times (8+6+9) + 3 \times (1+3) = 58$ bits, which saves $(81-58)/81 = 28.4\%$ space. We may see that the key of compression is to allocate shorter codes for more frequent characters.

char	code
a	10
b	01
c	000
d	001
e	11

Table 2: Example Huffman Coding

Before LZSS, we would briefly illustrate its predecessor, LZ77, which is a lossless compression algorithm named after the author A. Lempel and J. Ziv as well as the publication year 1977 [10]. It maintains a dictionary of a string buffer, and compresses the string data by representing the sub-strings as references to that in the dictionary. This dictionary slides as encoding and decoding, and is thus called a *sliding window*. The references is encoded as a length-distance pair (l, d), which means the following string of length l is exactly the same as the characters d distance behind in the uncompressed stream.

However, in this design, the authors suggests all strings be encoded in the dictionary, even though it has no matches in the dictionary. A reference to 1 or 2 characters occupies more bits than the original string, and this might end up in cases where a processed data is even longer than the original one [4]. To solve this, in 1982, James A. Storer and Thomas Szymanski proposed omitting such cases for better compression, and that scheme is named as LZSS [8]. Besides, to indicate the difference between literals and dictionary references, LZSS also introduces a 1-bit flag to label the following data chunks. The simplified scheme is shown in Algorithm 4 [4].

Algorithm 4: LZSS

Input : Unencoded input
Output: Encoded output

Initialize dictionary D `/* Already gotten a dictionary anyway */`
$output \leftarrow$ empty
$string \leftarrow$ part of uncoded string of maximum match length allowed
while *input is not entirely encoded* **do**
  search in $string$ for longest matching string in D
  if *match found* m *And* m *longer than reference* **then**
    $output.add$ encoded flag $True$
    $output.add$ distance and length (d, l)
  else
    $output.add$ encoded flag False
    $output.add$ first uncoded symbol $string[1]$
  end if
  Copy symbols newly written encoded output to D
  Shift $string$ by length of newly written encoded output
end while

Decoding

The process of decoding is simply a reversion of the encoding, but to completely read a PNG file, we would need specifically designed decoders. They should first read the image head and critical data chunks which indicate the encoding details. A PNG file starts with an 8-byte signature `IHDR` as the Table 3 specifies [9].

After the header there comes a series of data chunks. A typical data chunk is made of 4 parts: 4-byte *length* the chunk length, 4-byte chunk type/name, *length*-byte chunk data and 4-byte cyclic redundancy code/checksum (CRC) for data authentication [9]. Chunks are divided in to critical ones and ancillary ones. The critical chunks store the major information of the image for decoding, including `IHDR` the file head, `PLTE` the palette/color list, `IDAT` the image (might be split among multiple chunks) and `IEND` all-0 bytes as the image end. On the other

Width	4 bytes	Height	4 bytes
Bit Depth	1 byte	Color Type	1 bytes
Compression method	1 byte	Filter method	1 byte
Interlace method	1 byte		

Table 3: PNG Image Head `IHDR`

hand, the ancillary chunks are not necessary and can be skipped if the decoder cannot understand. It involves other image information like transparency, color space, histogram, text information, time stamp and so on.

After the decoder understand the detailed ways of filtering and compression, it is only a reverse of encoding to decode the PNG data. The decoding of LZSS result is easy by using the same dictionary and decide whether to transform the string by checking the encoded flag. If encoded flag is `True`, then look up the dictionary and copy the string referred to to the current position. The Huffman encoded bit-string can also be directly recovered to the original string by looking up the codes. While filtering is just a linear combinations of the current and neighbouring pixels, and can be restored by re-calculating the bytes from the last scan line back to the first.

Characters of PNG

One of the main advantages of PNG format is the minimum compression loss and the portability. Even if the size of the image is made to be small, the image quality is not damaged. Another major compressed image file format JPEG can also achieve good compression effects, but as a lossy compression method, some of the information would be definitely lost. And compared with its ascender GIF, PNG shows better compression ability in most cases.

Another point is that PNG support more color as well as transparency. As in GIF and JPEG only 8-bit indexed color are allowed, while PNG offers more choices, including 8 or 16-bit per channel true color. The alpha channel in PNG also makes transparent images possible.

PNG also supports addition of meta-file, that is, to add whatever text information you like to the file. It is a common practice in digital cameras and smart phones to add time stamp, location, shooting parameters, author and even copyrights. This function would be useful in some cases.

References.

[1] Thomas Boutell and T Lane. "Png (portable network graphics) specification version 1.0". In: *Network Working Group* (1997), pp. 1–102 (cit. on p. 1).

[2] Lee Daniel Crocker. "PNG: The portable network graphic format". In: *Dr Dobb's Journal-Software Tools for the Professional Programmer* 20.7 (1995), pp. 36–45 (cit. on p. 2).

[3] Peter Deutsch. *RFC1951: DEFLATE compressed data format specification version 1.3*. 1996 (cit. on p. 2).

[4] Michael Dipperstein. *LZSS (LZ77) Discussion and Implementation*. Mar. 2015 (cit. on p. 4).

[5] David A Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101 (cit. on p. 2).

[6] Colt McAnlis. *How PNG Works*. medium.com. Apr. 2016. URL: https://medium.com/@duhroach/how-png-works-f1174e3cc7b7 (cit. on p. 1).

[7] Alan W. Paeth. *Image file compression made easy*. Dec. 1991 (cit. on p. 1).

[8] James A Storer and Thomas G Szymanski. "Data compression via textual substitution". In: *Journal of the ACM (JACM)* 29.4 (1982), pp. 928–951 (cit. on p. 4).

[9] W3C. *Portable Network Graphics (PNG) Specification (Second Edition)*. Nov. 2003. URL: https://www.w3.org/TR/PNG/ (cit. on pp. 1, 4).

[10] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343 (cit. on p. 4).