WTF is wtf?

June 13, 2013

1 What is it?

Wtf is a file format that contains a picture compressed with **wavelet transform. Below is explained what is a wavelet transform, how it can be applied to data compression, and what is the structure of a wtf-file.**

Notice that this is only schoolwork, so wtf isn't very good compression. To be more exact, it is very bad and even lossy wtf-file is worse than lossless png.

2 What is a wavelet transform?

Wavelet transform is a way to express a function as a series of other simpler functions. It resembles the Fourier transform a lot. Here it is explained very briefly through an example. If you want to know more, check some of the references.

The space of all square integrable functions

$$L^2 := \{f : [0,1] \to \mathbb{R} : \int |f| < \infty\}$$

is a Hilbert space, with the inner product

$$< f, g >:= \int fg$$

and the corresponding norm:

$$\|f\|_2 := \sqrt{\int f}.$$

What we want to do is to have basis in $L^2[0,1]$, i.e. a set of fuctions we can express all the other functions with. The basis is better if it's orthogonal: $< u, v >= 0$ for every $u, v \in L^2[0,1] : u \neq v$. It is even better if all the functions in the basis have norm 1.

In order to construct the basis, let's define a function

$$w_0 : [0,1] \to \mathbb{R} : x \mapsto \begin{cases} 1, \text{ when } x < 1/2 \\ -1 \text{ otherwise.} \end{cases}$$

This is the mother wavelet from which the all the other wavelets (except one) will be formed with translations.

What we will do is divide the interval into half, and scale w_0 to the left half for function $w_{1,0}$ and to the right half for function $w_{1,2}$:

$$w_{1,0} : [0,1] \to \mathbb{R} : x \mapsto \begin{cases} 1, \text{ if } x < 1/4 \\ -1, \text{ if } 1/4 \leq x < 1/2, \\ 0, \text{ otherwise} \end{cases}$$

and

$$w_{1,1} : [0,1] \to \mathbb{R} : x \mapsto \begin{cases} 1, \text{ if } 1/2 \leq x < 3/4 \\ -1, \text{ if } 3/4 < x \\ 0 \text{ otherwise.} \end{cases}$$

The wavelets $w_{2.0}, \ldots, w_{2.3}$ are made in a similar fashion by splitting the intervals $[0, 1/2]$ and $[1/2, 1]$ in half and translating. From them we get the wavelets of the third level and so on.

Now, let's define one more function, and then we'll have the basis we were looking for. The function is simply 1 everywhere:

$$w : [0,1] \to \mathbb{R} : x \mapsto 1.$$

We'll call the set of all these functions W:

$$W := \{w, w_0, w_{k,j} : k = 1, 2, \ldots, j < k\}.$$

These, the wavelets w_0, $w_{k,j}$ and w (the characteristic function of the interval $[0, 1]$) make up an orthogonal basis for $L^2[0, 1]$. That means:

1. **Every function in $L^2[0, 1]$ can be represented as a weighted sum of the functions in W uniquely: For every function $f \in L^2[0, 1]$ there are unique constants c, c_0 and $c_{k,j}$ such that**

$$f(x) = cw(x) + c_0 w_0(x) + \sum_{k,j} c_{k,j} w_{k,j}(x).$$

2. **The set W is orthogonal, i.e. $< u, v >= 0$ for every $u, v \in W : u \neq v$.**

Things will become easier, when the wawelets are normalized, so let's redefine them as

$$\frac{w_{k,j}}{\|w_{k,j}\|_2}.$$

Now all the functions in W have the norm one, and thus W is orthonormal. The benefit of this is that if we represent a function $f \in L^2[0, 1]$ in this orthonormal base, the coefficients of the transfrom are very easy to calculate:

$$f(x) = \sum_{v \in W} < v, f > v(x).$$

Normalizing the wavelets changes only their nonzero values. The wavelets $w_{1,0}$ and $w_{1,2}$ for example have values $\sqrt{2}$ and $-\sqrt{2}$ where they

used to be 1 and -1, nothing else changes. The complete and final description of the wavelets $w_{k,j}$ is thus:

$$w_{k,j} : [0,1] \to \mathbb{R} : x \mapsto 2^{k/2} w_0(2^k x - j).$$

(Here you can extend the mother wavelet to be zero in $\mathbb{R} \setminus [0,1]$).

You can create many kinds of wavelet basis suitable to your needs by selecting carefully the mother wavelet. Haar wavelet is the easiest, and the most illustrative on the idea. It is not the most efficient for picture compression though. To keep language simpler, we'll call Haar wavelet transform in the rest of this just "wavelet transform".

The advantage wavelet transform has compared to the Fourier transform is the locality of it: suppose you have a function that is zero outside a small interval. If you perform a wavelet transform on it, lot of the coefficients will automatically be zeros. With Fourier transform you don't have this benefit.

3 How they can be applied to image processing?

Image on computer screen is a two dimensional array of colors. We assume here that the colors consist of three bytes, each of which descripe a shade of a basic color, red, green or blue. Thus an image can be thought of as a three dimensional array

`byte[][][] pic` where `pic[c][x][y]` tells what is the shade of the basic color `c` at the coordinates (x, y) of the picture.

It would be smart to introduce two dimensional wavelets to handle pictures, but wtf isn't that clever. Instead we do ordinary transfrom on single lines of data. The lines

`pic[c][x][0], pic[c][x][1]. pic[c][x][2], ...., pic[c][x][n]`

are transformed separately for each `c` and `x`.

If the line's length isn't some power of 2, it will be split to such lines and the transform is performed separately for each of them. So, we can now assume that the line's length is 2^n for some $n = 1, 2, \ldots$. To simplify the language, let's call one such line

$$\texttt{d[0], d[1], d[2],} \ldots \texttt{d[}2^n\texttt{]}).$$

We can use the ideas of the previous section, if we think the line as a function

$$f : [0,1] \to \mathbb{R} : t \mapsto \begin{cases} \texttt{d[0]} \text{ when} & t \in [0,\ 2^{-n}[\\ \texttt{d[1]} \text{ when} & t \in [2^{-n},\ 2 \cdot 2^{-n}[\\ \texttt{d[2]} \text{ when} & t \in [2 \cdot 2^{-n},\ 3 \cdot 2^{-n}[\\ \vdots & \\ \texttt{d[}2^n\texttt{-1]} \text{ when} & t \in [(2^n - 1) \cdot 2^{-n},\ 1] \end{cases}$$

If we perform a wavelet transform to it, the wavelets of level n and higher will produce zero coefficients. That's because they are nonzero only on intervals where f is constant, so the positive and

negative parts of them will negate each others. The finest level on which the wavelet produce nonzero coefficients is $n-1$.

Instead of calculating the wavelet coefficients, let us proceed another way that will make the computations easier. The normalization was just a theoretical background that will be used later. Right now we want to forget it, since it would produce irrational values and the calculations would be tedious.

Instead, we calculate coefficients like this: *Level* $n-1$, the coefficients are

$$\texttt{d[0]-d[1]}, \texttt{d[2]-d[3]}, \texttt{d[4]-d[5]}, \ldots, \texttt{d[}2^n\texttt{-1]-d[}2n\texttt{]},$$

Level $n-2$***:***

$$\begin{aligned}&\texttt{d[0]+d[1]-d[2]-d[3]},\\ &\texttt{d[4]+d[5]-d[6]-d[7]},\ldots\\ &,\texttt{d[}2^n\texttt{-3]+d[}2^n\texttt{-2] - d[}2^n\texttt{-1] - d[}2^n\texttt{].}\end{aligned}$$

And so on, until finally the *level* 1 has:

$$\texttt{d[0]+d[1]+}\ldots\texttt{+d[}2^{n-1}\texttt{] -d[}2^{n-1}\texttt{+1] - d[}2^{n-1}\texttt{+2]-}\ldots\texttt{-d[}2^n\texttt{].}$$

And then there's the sum of all the elemets `d[i]`, corresponding to the wavelet w.

These are *almost* the same as the coefficients $< w_{k,j}, f >$ of the previous section, but there are two differences:

1. **The wavelets aren't normalized.**
2. **These aren't exactly the inner product. If they were, every summed `d[i]` would have weight 2^{-n}.**

Therefore, if we call these coefficients b, b_0 and $b_{k,j}$ and the ones produced by the orthonormal wavelet transform c, c_0 and $c_{k.j}$, we have the relation:

$$c_{k,j} = 2^{k/2} \cdot 2^{-n} b_{k,j}. \tag{1}$$

Here the $2^{k/2}$ comes from normalization and 2^{-n} because we didn't weight the sum when calculating $b_{k,j}$:s. Similarly we have $c = 2^{-n}b$ and $c_0 = 2^{-n}b_0$,

Now we have the transform of the line of data. We can think of it as an array starting with the coefficients of the finest level and ending to the most robust ones:

$$(b_{n-1,0}, b_{n-1,1}, \ldots, b_{n-1,2^{n-1}-1}, b_{n-2,0}, \ldots b_{n-2,2^{n-2}-1}, \ldots, b_{1,0}, b_{1,1}, b_0, b).$$

This is essentially how the information is saved in the wtf-file too.

How wtf uses this transformati to pack the data is to forget the finest levels. It depends on the picture how much of it can be omitted, a screenshot with text in it is illegible with just two levels omitted.

To compute the transform with computer, we need the following simple method `sumTree`. It takes an array `data[]` and creates a binary tree out of it. The tree's leaves are the entries of `data[]`, their parents

are their pairwise sums ets., and the root is the sum of all the `data[]`. The tree is represented as an array with leaves as first entries and the root as the last. In our pseudocode the indices of the arrays start from 0.

sumTree
Input: `data[]`**, an array of length** 2^n**.**
Output: `sumTree[]`**, the tree as an array,**

1. **Create array** `sumTree[]` **with the length** `2*data.length -1`**.**
2. **For** `i =1` **to** `data.length -1` **do** `sumTree[i] = data[i]`**.**
3. **Set** `pointer = data.length`**.**
4. **Set** `levelLength = data.length / 2`**.**
5. **While** `(levelLength >= 1` **do 6 - 8**
6. **For** `i=0` **to** `levelLength -1` **do:**
   `sumTree[pointer+i] = sumTree[pointer - 2 * levelLength +2i] + sumTree[pointer -2 * levelLength + 2i +1]`
7. `pointer += levelLength`**.**
8. `levelLength /= 2`**.**

Here we do one level at time. The variable `pointer` **tells where it starts in the sum array, and** `levelLength` **tells how many entries there is in the level. The result array has almost twice the size of the original, and just a simple sum is calculated for each entry in it, so the time requirement is** $O(n)$**.**

The transform is calculated like this:

Input: `data[]`**, an array of data with the length** 2^n**.**
Output: `tr[]`**, the transform of** `data[]`**.**

1. **Calculate** `sumTree[]` **for** `data[]`**.**
2. **Create array** `tr[]` **with length** `data.length`**.**
3. **For** `i=0` **to** `tr.length -1` **do:** `tr[i] = sumTree[2i] - sumTree[2i+1]`**.**
4. `tr[tr.length -1] = sumTree[sumTree.length -1]`**.**

   The calculation of the sum tree takes $O(n)$ **and then each entry in the transform is a simple sum, so the total time requirement is** $O(n)$**.**

Now, the next objective is reconstructing the picture from this transformed data. Lets suppose that we have all the data, the lossy case is similar. We make use of the knowledge that

$$f(t) = \sum_{v \in W} < v, f > v(t) = cw(t) + c_0 w_0(t) + \sum_{k.j} c_{k,j} w_{k,j}(t),$$

where W is the set of normalized wavelets. From that and the relation (1) we get

$$f(t) = 2^{-n}\left(b\phi(t) + b_0\phi_0(t) + \sum_{k,j} 2^{k/2}\cdot 2^{k/2}\phi_{k,j}(t)\right)$$
$$= 2^{-n}\left(b\phi(t) + b_0\phi_0(t) + \sum_{k,j} 2^{k}\phi_{k,j}(t)\right),$$

where the functions ϕ, ϕ_0 and $\phi_{k,j}$ are the corresponding unnormalized wavelets. For the functions $w_{k,j}$ and $\phi_{k,j}$ applies

$$w_{k,j} = \frac{\phi_{k,j}}{\|\phi_{k,j}\|_2} = 2^{k/2}\phi_{k,j},$$

and with norm 1 the functions ϕ and ϕ_0 are equal to w and w_0.

That's all there is to it. When doing the computations with a machine, you can follow the following algorithm:

Input: `tr[]`**, an array of transform data.**
Output: `data[]`**, the original data retrieved from the transform data.**
The steps:

1. **Create the array** `data[]` **equal to the length of** `tr[]`**.**
2. **Set** `pointer = 0`
3. **Set** `coeffs = tr.length / 2`
4. **While** `(coeffs >= 1)` **repeat 5 - 7**
5. `add(data, tr, pointer, coeffs)`
6. `coeffs = coeffs /2`
7. `pointer = pointer + coeffs.`
8. **for each** `data[i]` **set** `data[i] = (data[i] + tr[tr.length -1] / tr.length`

We simply go through each level of coefficients and apply them to the data with the add-function, which is below. At the last line we add the last of the coefficients (the sum of all data) and divide by the number of data points, which gets it's explanation from the equation 1.

Input: `data[], tr[], pointer, coeffs.`

1. `stepLength = data.length / (2 * coeffs)`
2. **for** `c =0` **to** `coeffs -1` **do 3 - 5**
3. **for** `i=0` **to** `stepLength -1` **do:**
   `data[c*2*stepLength +i] += tr[c] * coeffs`

4. **for** `i=0` **to** `stepLength -1` **do:**
   `data[c*2*stepLength +stepLength + i] -= tr[c] * coeffs`

5. `c++`.

What happens here is that we take each coefficient at time, first add and then substract. The multiplier `coeffs` comes from the equation (1).

To calculate the time requirement, notice that for a data or transformation array of n entries there are $\log n$ levels of wavelets. For each level something is summed to each of the entries in the data array, so it takes $O(n \log n)$ time.

***The lossy case* is done almost like the lossless. We just omit one or more levels of the wavelets starting from the finest. The transform is similar, except the leaves of the sum tree are sums of $2^{\texttt{lol}}$ entries of the data array, where `lol` is the level of loss. The time requirement doesn't change, since you still have to sum all the entries.**

In the inverse transfrom you do everything the same way, except in the last line you don't divide by the size of the transform array, but by the size of the original data array instead. That size should be retrieved from other source than the forumla `tr.length * 2`$^{\texttt{lol}}$, since we want the transforms to have minimum size of 1 even when the level of loss would make it smaller.

The time requirement for lossy inverse transform is $n(\log n - \texttt{lol})$, given that the loss isn't too big.

4 The structure of a wtf file

This is the content of the header of a wtf file:

Offset	type	content
0	byte	File type
1	short	Number of colors
3	int	Original width
7	int	Original height
11	short	Level of loss
13	int	Compressed height

The offsets are bytes from the start of the file.

***File type* tells how the actual data is coded. There are two types: Integer data and mixed data, which are marked with 0 and 1 respectively.**

***Number of colors* tells how many colors the picture is comprised of.**

***Original width and height* tell the dimensions of the original picture.**

***Level of loss* tells how many levels of coefficients were omitted in the transform.**

***Compressed height* tells how long is the line of compressed data.**

This structure allows some other data to be packed with wtf too, although there are limitations.

The actual data starts at the offset 17. Regardless of the type of the file, the numbers are generated like this:

1. **A line is divided into pieces so, that the length of each piece is the biggest power of two you can fit into the remaining line. For example, a line of length 1500 would be split into lines of lengths 1024, 256, 128, 64, 16, 8 and 4.**

2. **Each of these sublines is transformed separately. The coefficients produced will be presented in the order described in the previous section (finest wavelets first).**

3. **The coefficients of each sublines follow each others so that the longest subline is first etc. These form a *compressed line* whose length was told in the header.**

4. **The compressed lines are written in the file so that they are separated first by the color and then by the column:** `color0`, `column0`, `color0`, `column1`, `color0`, `column2`, ... `color0`, `columnN`, `color1`, `column0`, `color1`, `column1`, ... `colorM`, `columnN`.

In the integer data type all the coefficients are written as integers. This takes awfully lot of space, and thus isn't used.

In the mixed data type each line of the compressed line contains two integers: `shortOffset` **and** `intOffset`**. Then in the line every number will be presented as bytes until the index** `shortOffset`**. After that the numbers are shorts until** `intOffset`**, and the rest of them are integers. (Java data types).**