DETECTING SIMILAR AND IDENTICAL IMAGES USING PERSEPTUAL HASHES

Couple of my hobbies are travelling and photography. I love to take pictures and experiment with photography. Usually after my trips, I just copy the photos to either my iPad or couple of my external hard disks. After 10 years, I have over 200K photos distributed across several disks and machines. I had to find a way to organize these photos and create a workflow for future maintenance. In this post I want to address one of the issues I had to solve: ** finding duplicate images **.

First, I needed to find out what exactly is a duplicate image. Analysing my photos, I found couple of interesting things:

1. Identical images: There were multiple copies of the same photo in different directories with different names.
2. Similar images: I usually bracket (exposure compensate or flash compensate) important pictures. So I have photos that visually appear to be the same, but may be a little darker/lighter based on exposure or flash settings.

Identical photos (1) are easy to find. These files are exactly the same so their bit patterns are the same. We could simply compute the checksum (or md5) of the photos and compare to find identical images.

Finding similar photos (2) is a little more challenging. The photos are visually the same, but they have enough differences that their checksum do not compare. Some of the differences that I found:

1. Exposure or flash compensated, so photos were slightly lighter/darker than the original.
2. A small difference in the point of focus or a slight change in point of view.
3. Photos with changed aspect ratio.
4. Resized photos, etc.

I had to find a way to detect all these changes to a photo and mark all these as duplicates of the original.

Researching "image similarity detection algorithms" on the web yields several techniques and tools. The primary theme of all these techniques was to calculate a fingerprint or hash of the image based on the perseptual content, not the raw bits in the image. A perceptual fingerprint/hash is derived from the visual features of the image. Unlike cryptographic hash functions like MD5 which rely on the fact that small changes in input leads to drastically different hash values, perceptual hashes produce hash values close to one another if the visual features of images are similar.

IMAGE SIMILARITY ALGORITHMS

There are several algorithms that can calculate an image fingerprint, some were based on heuristics while others had a solid mathematical backing. Here is a rundown of some of the techniques and tools I came across while researching this topic.

COLOR HISTOGRAM AS FINGERPRINT

The simplest of these fingerprinting techniques is using color histograms. Essentially a color histogram will capture the color distribution of the image. By comparing the normalized color histogram of images we can see if the color distributions match.

This techniques is apparently used in image retrieval/matching systems and are a standard way of matching images that is very reliable, relatively fast and very easy to implement. This type of matching is pretty resiliant to scaling (once the histogram is normalised), and rotation/shifting/movement etc.

Here is an implementation of an histogram indexing as described in Indexing via colour histograms - Swain, Ballard - 1990. See the table below to compare color histogram based similarity with other techniques.

GQVIEW'S IMAGE COMPARISON

GQview is an image viewer on Linux. It was one of the earliest programs that supported image similarity detection. Reading the code, this is what it does to detect similar images:

1. Divide the image into a 32 x 32 grid (1024 rectangles)
2. For each of the rectangle in the grid, average the red, green & blue color channels and store in 3 seperate 32 x 32 array. These 32 x 32 arrays represents the signature of the image.
3. To compare two images, compute the signatures of both the images and calculate the average of the corresponding array differences. i.e. similarity = 1 - (abs(red[img1] - red[img2]) + abs(blue[img1] - blue[img2]) + abs(green[img1] - green[img2])) / 255 * 1024 * 3

   1.0 is considered an exact match, while 0.0 for exact opposite images. Generally only a match of > 0.85 are significant at all, and >.95 is useful to find images that have been re-saved to other formats, dimensions, or compression.

FINDIMAGEDUPES.PL'S ALGORITHM

A pearl script written by rob kudla in 2001. It basically creates a fingerprint of the photos using Image Magick after applying a series of reductions. From the code, this is what findimagedupes does:

1. standardize image size by resampling to 160x160.
2. grayscale it.
3. blur it a lot. (gets rid of noise)
4. normalize (spreads the intensity out as much as possible)
5. equalize (make it as contrasty as possible): this is for those real dark pictures that someone has slapped a pure white logo on.
6. resample down to 16x16.
7. reduce to 1bpp: This basically uses a threshold to convert each pixel to either a white (0) or black (1) to 1 bit value.
8. convert to raw mono
9. Get the first 32 bytes of the raw image data. This is basically the fingerprint of the image.

When comparing images, findimagedups uses each images fingerprint and XORs them. The count of 1 bits in the result gives an approximate similarity score.

USING AVERAGE HASHING

This method is described by Dr. Neal Krawetz on the HackerFactor blog. The basic idea was to filter out high frequencies in an image and just keep the low frequencies. With pictures, high frequencies give you detail, while low frequencies shows the structure. A large, detailed picture has lots of high frequencies. A very small picture lacks details, so it is all low frequencies.

Here is the algorithm Dr. Neal described:

1. resize to a common 8x8 size. The fastest way to remove high frequencies and detail is to shrink the image.
2. grayscale it. This changes the hash from 64 pixels (64 red, 64 green, and 64 blue) to 64 total colors.
3. Compute the mean value of the 64 colors. This is the averaging of the hash.
4. Convert the 64 colors to 64 bits. Each bit is simply set based on whether the color value is above or below the mean.
5. Construct the hash. Set the 64 bits into a 64-bit integer. The order does not matter, just as long as you are consistent.
6. To compare two images, calculate the Hamming distance between two average hashes. A distance of zero indicates that it is likely a very similar picture (or a variation of the same picture). A distance of 5 means a few things may be different, but they are probably still close enough to be similar. But a distance of 10 or more? That's probably a very different picture.

According to Dr. Neal, the resulting hash won't change if the image is scaled or the aspect ratio changes. Increasing or decreasing the brightness or contrast, or even altering the colors won't dramatically change the hash value.

Here is an implementation of average hash using CImg library in C++. The similarity is computed as 1 - distance/64, so 1.0 means the images are similar. See the results of Average hash in the table below.

USING PHASH

According to Dr. Neal, a better approach is using pHash. Here is how pHash is computed:

1. Reduced to grayscale.
2. Resize the image to 32x32.
3. Compute the Discrete Cosine Transform (DCT) of the image. The DCT separates the image into a collection of frequencies and scalars.
4. Just keep the top-left 8x8 of the DCT. While the DCT is 32x32, the top-left 8x8 represents the lowest frequencies in the picture.
5. Compute the median value.
6. Compute the hash from the DCT. Set the 64 hash bits to 0 or 1 depending on whether each of the 64 DCT values is above or below the median value.

pHash is the most promising of all (see table below). There is an opensource version of pHash library that can be easily used in Ruby.

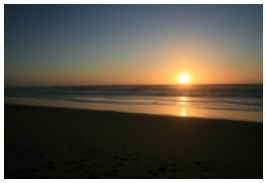
MULTI-RESOLUTION HAAR WAVELET DECOMPOSITION

ImgSeek is an opensource tool that computes image features based on an algorithm described in the paper Fast Multiresolution Image Querying. These features consist of 41 numbers for each colour channel (the code currently works in the YIQ colourspace).

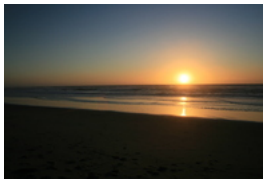
40 of these numbers will correspond to the 40 most significant wavelets found in a Haar wavelet decomposition of the image. The final number is based on the average luminosity of the image, and is basically a compensation factor. The image similarity is given by the sum of the weights for the most significant wavelet features, minus a component based on the average luminosity.

RESULTS

The table below shows the similarity scores for the following 4 images compared to the original.

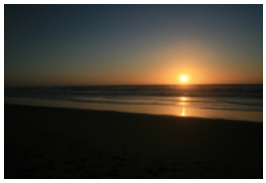


Original Image



Duplicate Image

Resized Image



Exposure Compensated Image

Cropped Image

	Original vs Duplicate	Original vs Resized	Original vs Exposure Compensated	Original vs Cropped
Color histogram intersection	1.0	0.00099	0.8104	0.00099
Average Hash	1.0	0.625	0.6875	0.6875
pHash	1.0	0.875	0.890	0.718
wavelet decomposition	1.0	0.99	0.886	0.298

Conclusion:

1. For the identical image (Original Vs Duplicate), Color histogram, Average Hash, pHash and imgseek returned 1.0, so they all can detect identical images.
2. For nearly identical images (Original Vs Exposure compensated) both imgseek and pHash gave a much better similarity score (~0.88) than Average hash (0.814). Color histogram did come very close at 0.81.
3. For scaled images (Original Vs Resized (90% of original)) imgseek was far better (0.99) than pHash (0.875) and Average Hash (0.625). I am not sure why Color histogram did rather poorly (0.00099). It could be something in my code.
4. For completely different images (Original vs Cropped), imgseek gave the biggest dis-similarity score (0.298) than both Average & pHash.
5. imgseek was the fastest of all the fingerprinting techniques.

Based on this, my choice was ImgSeek or pHash. I would have preferred to use imgseek, but I decided to use pHash as there was an open source implementation of pHash that can be easily used in a ruby script.

COMPARING PHASHES OF 200K IMAGES

We know we can easily compute the hamming distance of two pHashes and see how similar they are. But when we have 200K images, we need to compare an image with a very large set of pHashes to compute similarity. Linear comparison is out of question. A hash can only give us exact matches or identical images but not similar images. We need a data structure that can find near matches to hashes within a threshold.

A stack overflow entry pointed me in the right direction.

Question: What do we know about the Hamming distance d(x,y)?

Answer:

1. It is non-negative: d(x,y) 0
2. It is only zero for identical inputs: d(x,y) = 0 x = y
3. It is symmetric: d(x,y) = d(y,x)
4. It obeys the triangle inequality, d(x,z) d(x,y) + d(y,z)

This means that the Hamming distance is a metric for a metric space. There are good algorithms and data structures for indexing metric spaces: Metric tree, BK-tree, M-tree, VP-tree, Cover tree, etc.

In fact the open source pHash library has an implementation of Multi Vantage Point (MVP) trees. I needed more control over the tree nodes, so I decided to use a Burkhard Keller tree (BK-tree) in Ruby.

For a really good introduction to BK Trees see Damn Cool Algorithms.

PHOTO ORGANIZER TOOL

Finally, I wrote a general purpose photo organizer tool that reads the exif data in an image and copies it to a folder structure based on the data-time when the photo was taken. It uses pHash for generating image signatures and skips identical images and marks similar images as "DUPLICATE_OF" the original image.

Check it out: https://github.com/HackerLabs/PhotoOrganizer.

PHOTO ORGANIZATION RUBY SCRIPTS PHASH AHASH HAAR WAVELETS DUPLICATE PHOTO DETECTION

COMMENTS