The Hacker Factor Blog

Learn from your mistakes (you will learn a lot today)

Kind of Like That

Monday, 21 January 2013

Over at FotoForensics, we're about to enable another porn filter. Basically, there are a small number of prohibited images that have been repeatedly uploaded. The current filters automatically handle the case where the picture has already been seen. When this happens, users are shown a prompt that basically says "This is a family friendly site. The picture you have uploaded will result in a three-month ban. Are you sure you want to continue? Yes/No" Selecting "No" puts them back on the upload page. About 20% of the users select "Yes" and are immediately banned. (I can't make this up...)

The problem is that images may vary a little. Since the hash used for indexing images on FotoForensics differs, the auto-ban filter never triggers. So I need a way to tell if the incoming image looks like a known-prohibited image.

My current solution is to generate a hash of each known prohibited content and then test every new picture against the hashes. If it matches, then the user will be prompted to confirm that the new picture isn't prohibited content before continuing.

While leaving the choice to the user may seem too trustworthy, the same type of prompting has significantly reduced the amount of porn from 4chan. Uploads from 4chan used to be 75% porn. With prompting, it has dropped to 15%. Prompting users really does work.

A Different Approach

About 8 months ago I wrote a blog entry on algorithms for comparing pictures. Basically, if you have a large database of pictures and want to find similar images, then you need an algorithm that generates a weighted comparison. In that blog entry, I described how two of the algorithms work:

- **aHash** (also called Average Hash or Mean Hash). This approach crushes the image into a grayscale 8x8 image and sets the 64 bits in the hash based on whether the pixel's value is greater than the average color for the image.

- **pHash** (also called "Perceptive Hash"). This algorithm is similar to aHash but use a discrete cosine transform (DCT) and compares based on frequencies rather than color values.

As a comment to the blog entry, David Oftedal suggested a third option that he called a "difference hash". It took me 6 months to get back to evaluating hash

About

Dr. Neal Krawetz writes The Hacker Factor Blog. Follow him on Mastodon.

Popular Posts

• How Not to do Image Analysis Part I and Part II
• Looks Like It
• Body By Victoria
• Direct Deposit, Direct Withdrawl
• Point-of-Sale Vulnerabilities

Tools

• FotoForensics: Test your own photos.
• Hintfo: View metadata.
• Gender Guesser: Use your words.

Links

Security
Internet Storm Center
Krebs on Security
Bruce Schneier
Tao Security

Images
Photo Stealers
Awkward Family Photos
Unsplash

Debunking News
iMediaEthics
Poynter

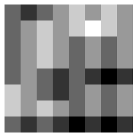
Debunking Politics
FactCheck
PolitiFact

Debunking Other
Snopes

functions and dHash is a definite winner.

dHash

Like aHash and pHash, dHash is pretty simple to implement and is far more accurate than it has any right to be. As an implementation, dHash is nearly identical to aHash but it performs much better. While aHash focuses on average values and pHash evaluates frequency patterns, dHash tracks gradients. Here's how the algorithm works, using the same Alyson Hannigan image as last time:

1. **Reduce size**. The fastest way to remove high frequencies and detail is to shrink the image. In this case, shrink it to 9x8 so that there are 72 total pixels. (I'll get to the "why" for the odd 9x8 size in a moment.) By ignoring the size and aspect ratio, this hash will match any similar picture regardless of how it is stretched.
2. **Reduce color**. Convert the image to a grayscale picture. This changes the hash from 72 pixels to a total of 72 colors. (For optimal performance, either reduce color before scaling or perform the scaling and color reduction at the same time.)
3. **Compute the difference**. The dHash algorithm works on the difference between adjacent pixels. This identifies the relative gradient direction. In this case, the 9 pixels per row yields 8 differences between adjacent pixels. Eight rows of eight differences becomes 64 bits.
4. **Assign bits**. Each bit is simply set based on whether the left pixel is brighter than the right pixel. The order does not matter, just as long as you are consistent. (I use a "1" to indicate that P[x] < P[x+1] and set the bits from left to right, top to bottom using big-endian.)
=  = 3a6c6565498da525

As with aHash, 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. Even complex adjustments like gamma corrections and color profiles won't impact

Math with Bad Drawings

Calendar

Archives

Feeds

Categories

the result. And best of all: this is FAST! Seriously -- the slowest part of the algorith
reduction step.

The hash values represent the relative change in brightness intensity. To compare
count the number of bits that are different. (This is the Hamming distance.) A valu
same hash and likely a similar picture. A value greater than 10 is likely a different i
between 1 and 10 is potentially a variation.

Speed and Accuracy

From FotoForensics, we now have a testbed of over 150,000 images. I have a coup
that occur a known number of times. For example, one picture (needle) appears ex
150,000 image repository (haystack). Another picture occurs twice. A third test pic
occurs 32 times.

I've used aHash, pHash, and dHash to search for the various needles in the haysta
comparisons, I did not pre-cache any of the repository hash values. I also consider
10 to denote a match or a miss. (If the haystack image differs from the needle ima
bits, then it is assumed to not match.) Here's the results so far:

- **No hash**. This is a baseline for comparison. It loads each image into memory
the image. This tells me how much time is spent just on the file access and lo
images are located on an NFS-mounted directory -- so this includes networl
The total time is 16 minutes. Without any image comparisons, there is a min
minutes needed just to load each image.

- **No hash, Scale**. Every one of these hash algorithms begins by scaling the im
pictures scale very quickly, but large pictures can take 10 seconds or more. J
scaling the 150,000 images takes 3.75 hours. (I really need to look into possi
optimizing my bilinear scaling algorithm.)

- **aHash**. This algorithm takes about 3.75 hours to run. In other words, it takes
and scale the image than to run the algorithm. Unfortunately, aHash genera
of false positives. It matched all of the expected images, but also matched al
false-positives. For example, the test picture that should have matched 32 ti
matched over 400 images. Worse: some of the misses had a difference of le
general, aHash is fast but not very accurate.

- **pHash**. This algorithm definitely performed the best with regards to accurac
positives, no false negatives, and every match has a score of 2 or less. I'm su
data set (or alternate needle image) will generate false matches, but the nur
likely be substantially less than aHash.

  The problem with pHash is the performance. It took over 7 hours to complet
the DCT computation uses lots of operations including cosine and sine. If I p
DCT constants, then this will drop 1-2 hours from the overall runtime. But ap
coefficients still takes time. pHash is accurate, but not very fast.

- **dHash**. Absolutely amazing... Very few false positives. For example, the imag
matches ended up matching 6 pictures total (4 false positives). The scores w
and 10. The two zeros were the correct matches; all of the false-positive ma

scores. As speed goes, dHash is as fast as aHash. Well, technically it is faster
need to compute the mean color value. The dHash algorithm has all the spe
very few false-positives.

Algorithm Variations

I have tried a few variations of the dHash algorithm. For example, David's initial pr
8x8 image and wrapped the last comparison (computing the pixel difference betw
for the 8th comparison). This actually performs a little worse than my 9x8 variatio
positives), but only by a little.

Storing values by row or column really doesn't make a difference. Computing both
hashes significantly reduces the number of false-positives and is comparable to p
accurate). So it maintains speed and gains accuracy as the cost of requiring 128 bit

I've also combined pHash with dHash. Basically, I use the really fast dHash as a fas
matches, then I compute the more expensive pHash value. This gives me all the sp
all the accuracy of pHash.

Finally, I realized that using dHash as a fast filter is good, but I don't need 64-bits
computation. My 16-bit dHash variant uses a 6x4 reduced image. This gives me 20
Ignoring the four corners yields a 16-bit hash -- and has the benefit of ignoring the
Instagram-like vignetting (corner darkening). If I have a million different images, t
about 15 images per 16-bit dHash. pHash can compare 15 images really quickly. At
I'm looking at about 15,258 image collisions and that still is a relatively small numl

I can even permit my 16-bit dHash to be sloppy; permitting any 1-bit change to ma
16-bit dHash would yield 17 possible dHash values to match. A million images sho
collisions, and a billion becomes about 260,000 collisions. At a billion images, it w
storing the 16-bit dHash, 64-bit dHash, and 64-bit pHash values. But a million ima
ahead just by pre-computing the 16-bit dHash and 64-bit pHash values.

Applied Comparisons

There are two things we want out of a perceptual hash algorithm: speed and accu
dHash with pHash, we get both. But even without pHash, dHash is still a significan
over aHash and without any notable speed difference.

Given a solid method for searching for images, we can start exploring other resear
example, we can begin doing searches for the most commonly uploaded image va
expecting memes and viral images to top the list) and better segmenting data for
might even be able to enhance some of our other research projects by targeting sp
images.

In the meantime, I'm really looking forward to getting the new porn filter online.

Read more about Forensics, FotoForensics, Image Analysis, Programming | Comments (43) | Direct Li

COMMENTS

#1 jim (Homepage) on 2013-01-22 14:51 (Reply)

Interesting post! You could generate a large set of test images by downloading vi
out frames. Knowing which video the frame came from also helps in the test anal
images score a match and they are from the same video then it is likely to be a tru

#2 Our Lady of Infinite Loops on 2013-01-24 23:21 (Reply)

Does it change the results at all if you swap the order of

1) Reduce size. The fastest way to remove high frequencies and detail is to shrink
case, shrink it to 9x8 so that there are 72 total pixels...
2) Reduce color. Convert the image to a grayscale picture.

? Because it seems to me it'd be quicker to reduce the size of a grayscale, so if drc
grayscale in the large photo isn't all that expensive timewise and if the accuracy i:
might reverse the order of these two.

I'm not an image analysis expert, of course.

#2.1 Dr. Neal Krawetz (Homepage) on 2013-01-25 08:08 (Reply)

Great question. There may be fractional numerical differences depending on h
scaling and gray conversions, but in all practical terms: no -- it doesn't make a
then-gray or gray-then-scale should give you the same gradient directions.

In fact, you can do the scaling and grayscale conversion in the same step if you
systems that are memory restrictive.)

As far as speed goes... Interesting question. Gotta go benchmark it!

Update: Just checked. My existing code does grayscale first and scaling seconc
order does run slower because there is 3x scaling for RGB images. However, tl
difference isn't noticed until you hit really large images. I'll update the blog en
optimal order of events.

#3 h1fra (Homepage) on 2013-01-26 19:40 (Reply)

Awesome post. Very interesting how this simple hack could be very efficient.
I wonder if we could see the code anywhere? 🙂

#4 olewang (Homepage) on 2013-02-28 00:40 (Reply)

if dhash is better than average hash,i want to try. Actully i use the ahash, the perf
good.

#5 student on 2013-05-15 07:04 (Reply)

Hi,
I have a question, if the perceptual hashing could also be used for text/plagiarism
I will try it. If no, what could be the algorithm for that?

Looking forward to receive an answer to my question.

Student

#6 mm on 2013-05-31 08:18 (Reply)

what about images with watermark, i think this will really mess up the dhash...

#7 marcan (Homepage) on 2013-08-09 14:07 (Reply)

I'm surprised at the claim that pHash is slow. I reimplemented pHash in about 15
numpy, and it benchmarks at 5534 hashes per second on my box (a Core i7 laptop
resize step (which dHash also needs, of course). I've been using it on video, and th
and resize is by far the slowest part of the pipeline.

pHash's DCT is really just an (8x32) **(32x32)** (32x8) matrix multiplication chain, a r
64 comparisons, which is very fast to compute (it could probably be reduced to al
too without significantly affecting the output).

On your 150k test image set, it should take about 27 seconds, a negligible amoun
compared to the 3.75 hours it took to scale the images. I suspect something is ter
about your implementation if it's 430 times slower than my Python one!

I use pHash to cross-match about 2000 frames of "dirty" video (possibly subtitlec
with compression artifacts) against 300k-1m frames of clean HD video. I pHash e
both halves (which takes about an hour or two in total - mostly the H) and store t
a very small and simple bit of code.

Then I compare every one of the 2000 frames against every one of the 300k-1m f
billion comparisons) and look for runs of frames with similar hashes (by counting
bits that differ and comparing it with two thresholds). This takes a few minutes w
optimized Python + Cython code.

If you're interested, the video hashing script is here:
http://www.marcansoft.com/paste/czgaGc7a.txt

#7.1 Dr. Neal Krawetz (Homepage) on 2013-08-10 08:54 (Reply)

Hi marcan,

I had to triple-check my code. I'm using a very efficient DCT implementation. (A
precomputed once. Just a set of double loops for multiplication and summatic
compute the constants once regardless of the number of pictures being hashe

I think the Python code you're using might be taking advantage of MMX, SSE, S
other hardware optimizations. My code doesn't use this (yet). With SIMD opera
loops could be reduced to a single loop. And with SSE2 or FPGA, it might even
matrix is small enough to completely fit into the SIMD registers. This would ea
200%-400% speed improvement for the forward-DCT operation.

#7.1.1 marcan (Homepage) on 2013-08-12 06:42 (Reply)

Even if numpy is using SIMD instructions, that doesn't explain the massive s
between your implementation and mine. SIMD might speed things up 200%

say, but not 43000%. The code uses floats, and you can fit four 32-bit float
register, at best.

Am I misunderstanding the way you ran your benchmark? I'm considering tl
hash 150k images. Obviously if you are re-hashing the entire haystack for e
reporting the total time for all needles then that's a factor that I'm not takir
(you didn't specify how many needles you were searching for).

I just changed the Python code to implement matrix multiplication naively i
also perform the median by sorting and taking the middle element, removi
the way Python floats work, this also means it is now using slower double-
instead of single-precision floats; no way around that unless I switch to ints
It is, of course, much slower, but still benchmarks at 677.69 hashes per sec
time of 221 seconds (3.6 minutes) for the 150k-image test set. Keep in mind
Python code now running inside the Python VM, with tons of overhead. A p
implementation without using SIMD would be much faster.

#7.1.2 Ty on 2016-06-25 17:35 (Reply)

I think you're using the standard method of computing the DCT instead of t
method.

The FastDCT computes all of the expensive math upfront and than uses ma
to compute the DCT, making it extremely quick to run.

My c# FastDCT implementation was able to compute 3000 hashes in 800n
seconds on your image set), where as the standard method was so slow I c
it once!

Anyway, thanks again for the great algorithm! It's helped me immensely!

#7.1.2.1 ngoc on 2016-10-19 15:54 (Reply)

Hi. Is there anyway you can share your implementation with FastDCT he

I'm also experiencing worse results with dHash implemented in C# and I
why, yet...

#7.1.2.1.1 marcan (Homepage) on 2017-01-21 09:21 (Reply)

We had an email exchange about this after my comment. The problem
doing the naive 2-D DCT calculation the brute force way, which is O(n
row-column decomposition with matrix multiplication, which is $O(n^3)$
a massive speed difference between the two.

I posted my Python implementation in my comment above. That's the
method. You should be able to reimplement it in whatever language y

---

#8 victor on 2013-08-15 11:41 (Reply)

Is there a java based dHash implemention?

#8.1 daoxin on 2013-11-06 01:59 (Reply)

python version:
http://home.no/rounin/programming.html

php version:
http://jax-work-archive.blogspot.com/2013/05/php-ahash-phash-dhash.html

#9 victor on 2013-08-15 15:04 (Reply)

Does this support scaled or rotated image comparison?

#9.1 Dr. Neal Krawetz (Homepage) on 2013-08-15 20:15 (Reply)

Hi victor,

Scaled: yes.
Rotated: no.

#10 VaL (Homepage) on 2013-11-10 06:36 (Reply)

Hi,
For one project I need to prevent adding duplicate images by users.
This is an implementation of dhash, phash and ahash in PHP:
https://github.com/valbok/leie/blob/master/classes/image/leieImage.php

also in python (uses openCV for DCT and SURF)
https://github.com/valbok/img.chk/blob/master/core/image.py

Unfortunately these hashes do not work with cropped images.

But it can be used to determine duplicates like
https://github.com/valbok/leie/blob/master/tests/image/img/madonna.jpg
and
https://github.com/valbok/leie/blob/master/tests/image/img/madonna-a.jpg

And the main feature of these hashes is that hamming distance can be calculated
by using BIT_COUNT.

For example:
SELECT image_id FROM image_hash WHERE BIT_COUNT(YOUR_HASH ^ hash)

Today looking at how to index SIFT and SURF or other descriptors to find croppe
still a question for me how to make searching in a list of keypoints or features.

#11 Will on 2014-01-20 22:35 (Reply)

As a variation to dHash you can compare the pixels of the 8x8 image in the order
matrix instead of adjacent pixels.

#12 AlexHackerFactor on 2014-11-04 04:16 (Reply)

Another variation is to use a space-filling curve and tie its beginning to its end. Th
hashed image to be 8x8.

Too, this turns out to be pretty good way to handle images that are big patterns -

the ocean and sky. The normal dhash for such images tends to be values like 0xff
0x0000ffffFFFFffff.

There is another way to handle near-zero and near-0xfff... dhash values: If an ima
zero or F's, first rotate the image, then if it still hashes bad, use just the red band,
band. In my code, I've also done this sort of thing if the (hash ^ (hash >> 32)) & 0x
zero hash. But, in around 12,000 images, these sorts of things aren't done but on
dhash is done in a space-filling curve order.

So far as orientation is concerned, rotating the image so that it's wider than high
a lot of images.

Also, when looking images up by hash, a Python app I wrote looks up all orientati
and picks the best one - the one closest to the haystack image in question.

#13 PhillyCoder on 2015-07-10 12:36 (Reply)

Hello - great blog series on this topic! I've attempted to express DHash in C# bas
AHash implementation found here: https://github.com/jforshee/ImageHashing

However my findings are not great. I'm actually seeing worse tolerance to
brightness/contrast/color and size changes in DHash than in AHash. I'm using Ph
small changes.

Similarity results (100% == perfect match):

Brightness increase 5%: AHash (98%), DHash (90%)
Contrast decrease 10%: AHash(98%), DHash(90%)
Resized 25% smaller: AHash(100%), DHash(96%)
Resized 50% larger: AHash(98%), DHash(95%)

Some other important things to note:

The similarity is expressed 0-100% via the following: return ((64 - BitCount(hash
/ 64.0;

Grayscaling is done via:

```
uint gray = (pixel & 0x00ff0000) >> 16;
gray += (pixel & 0x0000ff00) >> 8;
gray += (pixel & 0x000000ff);
gray /= 12;

return (byte)gray;
```

The DHash implementation is:

```
ulong hash = 0;
for (int i = 0, j = 1; i < 72; i++, j++)
{
if (j > 1 && j % 9 == 0) // don't calculate the current end of row to the beginning o
continue;
if (grayscale[i] > grayscale[i + 1]) // greater than the next pixel? then 1 else 0
hash |= (1UL &lt;&lt; (63 - i));
}
```

I believe you guys are far wiser, so I believe your findings, however I cannot find t
implementation. Does anything jump out at you? Thanks in advance!

#13.1 Dr. Neal Krawetz (Homepage) on 2015-07-10 13:16 (Reply)

How are you scaling down the image to 9x8?

A subsample scale algorithm could cause this. But something like bilinear shou
way.

#13.1.1 PhillyCoder on 2015-07-11 08:01 (Reply)

Thanks for the quick reply! The code I extended from the AHash implement
standard .Net imaging routines:

```
// Squeeze the image into an 9x8 canvas
Bitmap squeezed = new Bitmap(9, 8, PixelFormat.Format32bppRgb);
Graphics canvas = Graphics.FromImage(squeezed);
canvas.CompositingQuality = CompositingQuality.HighQuality;
canvas.InterpolationMode = InterpolationMode.HighQualityBilinear;
canvas.SmoothingMode = SmoothingMode.HighQuality;
canvas.DrawImage(image, 0, 0, 9, 8);
```

In this case, the "image" variable in the last line is the high-res version, and
the canvas using the properties/qualities as outlined above. I would think th
InterpolationMode being HighQualityBilinear would suffice, perhaps not?

#13.1.1.1 PhillyCoder on 2015-07-11 14:11 (Reply)

Here are the results from tests I ran using different resizing methods. Th
the built-in .Net functionality, followed by the commercial AForge librar
perform well as you'll see), and finally a a bilinear algorithm I found onlir

For my application, I need the images to perform well with slight brightr
differences, I won't run into resizing but ran tests for that anyway. The w
HighQualityBicubic resampling. The similarity isn't as high as I would like
a 99-100% match) but maybe I can live with this.

.Net Interpolation Mode: HighQualityBilinear

Brightness increase 5%: AHash (98%), DHash (90%)
Contrast decrease 10%: AHash(98%), DHash(90%)
Resized 25% smaller: AHash(100%), DHash(96%)
Resized 50% larger: AHash(98%), DHash(95%)

.Net Interpolation Mode: HighQualityBicubic

Brightness increase 5%: DHash (96%)

Contrast decrease 10%: DHash(92%)
Resized 25% smaller: DHash(98%)
Resized 50% larger: DHash(95%)

AForge Lib Bilinear Resizing:

Brightness increase 5%: DHash (89%)
Contrast decrease 10%: DHash(85%)
Resized 25% smaller: DHash(93%)
Resized 50% larger: DHash(96%)

AForge Lib Bicubic Resizing:

Brightness increase 5%: DHash (92%)
Contrast decrease 10%: DHash(92%)
Resized 25% smaller: DHash(92%)
Resized 50% larger: DHash(89%)

LinearInterpolationScale C# Resizing:

Brightness increase 5%: DHash (90%)
Contrast decrease 10%: DHash(87%)
Resized 25% smaller: DHash(100%)
Resized 50% larger: DHash(100%)

#13.2 Kevin on 2015-07-17 14:05 (Reply)

The first concern that jumps out at me is the grayscaling. I'm guessing you're g
average", which would be:

(R + G + B) / 3

In your post you state you are dividing by 12, so all your colors have been reduc

http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grays

You might see what happens if you let .NET do the work for you by using:
Bitmap squeezed = new Bitmap(9, 8, PixelFormat.Format16bppGrayScale);

Kevin

#13.2.1 PhillyCoder on 2015-07-21 12:51 (Reply)

Thanks, Kevin. I borrowed that conversion code so I inherited the math prob
but AHash still returns a better similarity score than DHash.

I tried the approach you mentioned with the 16bpp Bitmap, but unfortunate
support Graphics.FromImage() for that pixel format (it's documented). I trie
RGB to Grayscale conversion but again, AHash is giving me better numbers
That algorithm is here: http://stackoverflow.com/questions/2265910/conve
grayscale

Scratching my head over this, but happy to stick with AHash if it gives me th
Strange though...

#14 Rick on 2015-09-28 21:06 (Reply)

Hi.

If we apply this to a large set of pictures is 8 pixels not too small so there will be a
and therefore false positives??

Is not the fingerprint more accurate if the hash_size value is larger??

Thank you

#14.1 Breakthrough (Homepage) on 2016-02-06 22:28 (Reply)

I'm really surprised nobody else commented on this issue. I'm sure you could e
image to fool the classifier, which raises the question of a false positive occurr
probability for an end user.

It would be worth-while investigating feature/edge detection (e.g. SIFT), as sca
and other factors, like stretching, flipping, rotating - is arguably one of the mo
problems when it comes to this sort of thing.

#14.1.1 Dr. Neal Krawetz (Homepage) on 2016-02-07 07:01 (Reply)

Hi Breakthrough,

Defeating detection is easy. Just alter about 20% of the image or change th
about 10% via cropping or adding.

However, the result is no longer a visually similar picture.

I have a coworker who calls these perceptual hashes "the 20 foot test" (or "
metric measurements). That is, if you print out both pictures and hang them
away, would you still think it's the same picture?

Different hash systems serve different purposes. These hashes look for visu
images. As long as you don't want your picture to be visually similar, you ca
so that it won't match the hash.

Regarding SIFT: SIFT is a different type of algorithm and solves a different s
(Unfortunately, SIFT is also patent restricted, and that dramatically limits th

---

#14.1.1.1 noob on 2021-06-09 03:07 (Reply)

Hi -

Thanks for a great article. I wonder if this "20 foot test" still work with ac
examples (generated to fool DNN classifiers). The adversarial examples
same from human perspective, yet it loos like many perceptual hashing
fail to detect the similarity.
Do you know if there is any research in this field? Thanks.

#14.1.1.1.1 JW on 2021-08-19 01:47 (Reply)

I don't think perceptual hashes would be fooled by adversarial examp
neural networks because those look at pixel detail, and just the right k
confuse them. But perceptual hashes like these smooth all noise away

Giving at try on the doggie from https://christophm.github.io/interpre
book/adversarial.html The ahash and phash are exactly the same, and
bit difference.

#14.1.1.1.1.1 Dr. Neal Krawetz (Homepage) on 2021-08-19 07:14 (Re

Hi JW,

It depends on the type of perceptual hash.

pHash and aHash are pretty robust, but a big enough picture can ha
changes that are removed during the downsampling (scaling small
look similar. (false-positive because of the 20-foot test)

PhotoDNA just needs a few specific edits performed in a verify spe
visually similar picture to have a completely different hash. (If you k
doing, you can alter as little as 0.6% of the picture and generate a f
negative.)

#15 David Oftedal (Homepage) on 2015-10-31 10:34 (Reply)

New locations for my implementations of average hash and dHash:

http://01101001.net/averagehash.php
http://01101001.net/AverageHash.py
http://01101001.net/Imghash.zip

http://01101001.net/differencehash.php
http://01101001.net/DifferenceHash.py

#16 Luca Lovagnini on 2016-05-21 00:43 (Reply)

Hello! Great article, really! There is any C++ implementation about dHash? I cann
anywhere!

#16.1 Stoney Vintson on 2017-12-20 12:53 (Reply)

openCV v3 has c++ hash implementations for average, perceptual, radial, Marr
moment. openCV does not have a difference hash, but it may be added. There
performance results for invariances on the ukbench data set.
https://docs.opencv.org/3.3.1/d4/d93/group__img__hash.html

#17 christoph on 2017-02-23 18:44 (Reply)

Instead of 9x8, I resize to 10x7, leading to 63 difference bits. This works slightly b
many images have a ~1.4 aspect ratio, so the feature extraction is distributed a bi
The 64th bit can optionally be used to mark images that had to be rotated to be v

#18 James McKenna on 2017-04-10 08:58 (Reply)

I use the variation of the algorithm wrapping the last comparison and using an 8x
doing this, you can also calculate the average hash with almost no added expense
hashes allows for a bigger hamming distance to be used which results in fewer fa
The added hash comparison also reduces false positives.

#19 Tom64b (Homepage) on 2017-05-12 13:42 (Reply)

Thank you Dr. Krawetz for the article!

If any of you need PHP code for dHash- here's my implementation in 50 lines (inc
comments):

https://github.com/Tom64b/dHash/blob/master/dhash.php

#20 danros on 2017-10-11 18:25 (Reply)

Is there a Java implementation. If not and will anyone be available to do a code re

#20.1 Kilian on 2018-08-20 03:19 (Reply)

Just saw your comment. I implemented it a while back.

https://github.com/KilianB/JImageHash

#21 Victor Maslov on 2020-08-02 08:51 (Reply)

Hello.

To make the fingerprinting faster in 2017 I had a look at existing dhash implemen
and made my own one powered by vips instead of imagemagick.
To increase the quality of the algorithm I've improved it in several aspects and ca
"IDHash".
To make the fingerprints comparison faster in 2019 I've implemented the functio
extension.

I always had benchmarks so currently I assume my gem to be the best Ruby impl
anyone is interested: https://github.com/Nakilon/dhash-vips

#22 Garrison on 2020-10-13 15:03 (Reply)

I've implemented the dhash for a "repost detecting bot", and something I've noti
in particular trigger it very often. The hash is failing because images that have dat
organized in lines (like a paragraph w/ lines of text) leads to the hash being 0, sin
dissimilarity between images.

I'm considering adding a secondary hash
- like phash - to cut down on repeats. I'm wondering if anyone else has run into si
if so how did you get around it?

Great article by the way!

#22.1 Diane (Homepage) on 2022-07-07 17:05 (Reply)

Use dHash - or the similar but more sensitive VisHash (https://github.com/GoI
LANL/VisHash) - as a first-pass. Then for the flagged matches or zero-signatu
another hash that is more sensitive and designed for that type of image (but li
computationally expensive). The paper linked in the GitHub README shows sc
of different hashes on different types of images (drawings and diagrams in pa
have different properties than photos. The AHDH hash may work well for you,
of a computationally efficient implementation.

ADD COMMENT

Code of conduct

- Name calling and anti-social comments will not be posted.
- Comments must be related to the topic. Unrelated comments will not be pos
are submitting your comment to the correct blog entry; Yes, people have sub
comments to the wrong blog entries.
- Comments should be rational and logical, citing findings as appropriate.
- Opinions and speculations are desired and welcome, but if they are represen
they may be moderated or censored.
- The moderator reserves the right to end tangential discussions and censor o
inappropriate content.

Name	
Email	
Homepage	
In reply to	[Top level]
Comment	

Enclosing asterisks marks text as bold (*word*), underscore are mad
Standard emoticons like :-) and ;-) are converted to images.
E-Mail addresses will not be displayed and will only be used for E-M

☐ Remember Information?

Submitted comments will be reviewed by moderators before being displayed.

Submit Comment Preview