Image comparison - fast algorithm

Asked 15 years, 7 months ago Modified 1 year, 11 months ago

Viewed 365k times

450

I'm looking to create a base table of images and then compare any new images against that to determine if the new image is an exact (or close) duplicate of the base.

For example: if you want to reduce storage of the same image 100's of times, you could store one copy of it and provide reference links to it. When a new image is entered you want to compare to an existing image to make sure it's not a duplicate ... ideas?

One idea of mine was to reduce to a small thumbnail and then randomly pick 100 pixel locations and compare.

image algorithm comparison computer-vision

Share

Improve this question

Follow

edited Aug 28, 2013 at 20:58

Chuck Savage

11.9k ● 6 ● 51 ● 69

asked May 9, 2009 at 20:18

meade

23.3k ● 12 ● 33 ● 36

12 Answers Sorted by: Highest score (default)

491

Below are three approaches to solving this problem (and there are many others).

- The first is a standard approach in computer vision, keypoint matching. This may require some background knowledge to implement, and can be slow.
- The second method uses only elementary image processing, and is potentially faster than the first approach, and is straightforward to implement. However, what it gains in understandability, it lacks in robustness -- matching fails on scaled, rotated, or discolored images.
- The third method is both fast and robust, but is potentially the hardest to implement.

Keypoint Matching

Better than picking 100 random points is picking 100 *important* points. Certain parts of an image have more information than others (particularly at edges and corners), and these are the ones you'll want to use for smart image matching. Google "keypoint extraction" and "keypoint matching" and you'll find quite a few academic papers on the subject. These days, SIFT keypoints are arguably the most popular, since they can match images under different scales, rotations, and lighting. Some SIFT implementations can be found here.

One downside to keypoint matching is the running time of a naive implementation: O(n^2m), where n is the number of keypoints in each image, and m is the number of images in the database. Some clever algorithms might find the closest match faster, like quadtrees or binary space partitioning.

Alternative solution: Histogram method

Another less robust but potentially faster solution is to build feature histograms for each image, and choose the image with the histogram closest to the input image's histogram. I implemented this as an undergrad, and we used 3 color histograms (red, green, and blue), and two texture histograms, direction and scale. I'll give the details below, but I should note that this only worked well for matching images VERY similar to the database images. Re-scaled, rotated, or discolored images can fail with this method, but small changes like cropping won't break the algorithm

Computing the color histograms is straightforward -- just pick the range for your histogram buckets, and for each range, tally the number of pixels with a color in that range. For example, consider the "green" histogram, and suppose we choose 4 buckets for our histogram: 0-63, 64-127, 128-191, and 192-255. Then for each pixel, we look at the green value, and add a tally to the appropriate bucket. When we're done tallying, we divide each bucket total by the number of pixels in the entire image to get a normalized histogram for the green channel.

For the texture direction histogram, we started by performing edge detection on the image. Each edge point has a normal vector pointing in the direction perpendicular to the edge. We quantized the normal vector's angle into one of 6 buckets between 0 and PI (since edges have 180-degree symmetry, we converted angles between -PI and 0 to be between 0 and PI). After tallying up the number of edge points in each direction, we have an un-normalized histogram representing texture direction, which we normalized by dividing each bucket by the total number of edge points in the image.

To compute the texture scale histogram, for each edge point, we measured the distance to the next-closest edge point with the same direction. For example, if edge point A has a direction of 45 degrees, the algorithm walks in that direction until it finds another edge point with a direction of 45 degrees (or within a reasonable deviation). After computing this distance for each edge point, we dump those values into a histogram and normalize it by dividing by the total number of edge points.

Now you have 5 histograms for each image. To compare two images, you take the absolute value of the difference between each histogram bucket, and then sum these values. For example, to compare images A and B, we would compute

```
|A.green_histogram.bucket_1 -
B.green_histogram.bucket_1|
```

for each bucket in the green histogram, and repeat for the other histograms, and then sum up all the results. The smaller the result, the better the match. Repeat for all images in the database, and the match with the smallest result wins. You'd probably want to have a threshold, above which the algorithm concludes that no match was found.

Third Choice - Keypoints + Decision Trees

A third approach that is probably much faster than the other two is using semantic texton forests (PDF). This involves extracting simple keypoints and using a collection decision trees to classify the image. This is faster than simple SIFT keypoint matching, because it avoids the costly matching process, and keypoints are much simpler than SIFT, so keypoint extraction is much faster. However, it preserves the SIFT method's invariance to rotation, scale, and lighting, an important feature that the histogram method lacked.

Update:

My mistake -- the Semantic Texton Forests paper isn't specifically about image matching, but rather region labeling. The original paper that does matching is this one: Keypoint Recognition using Randomized Trees. Also, the papers below continue to develop the ideas and represent the state of the art (c. 2010):

- Fast Keypoint Recognition using Random Ferns - faster and more scalable than Lepetit 06
- ~~BRIEF: Binary Robust Independent Elementary Features~~ - less robust but very fast -- I think the goal here is real-time matching on smart phones and other handhelds

Share Improve this answer

Follow

edited Apr 15, 2020 at 3:34

Wug

13.2k ● 5 ● 35 ● 56

answered May 9, 2009 at 21:27

Kyle Simek

9,636 ● 4 ● 27 ● 33

The Histogram approach seems to make the most sense. I'm assuming you can rotate the image to perform this on all sides just in case the image being compared to was turned (treating the same image as 4) - thanks – meade May 9, 2009 at 23:03

4 @meade That's right. Something else to consider: depending on your problem, you might not need to use all 5 histograms in your algorithm. Discarding the texture direction histogram will allow you to match rotated versions of the picture. Discarding the texture scale histogram will allow you to match re-scaled versions of the image. You'll lose some ability to compare similarity, but this might not be a problem, depending on your situation. Also, since computing texture information is the most costly part of the algorithm, this will make your algorithm speedy, too. – Kyle Simek May 9, 2009 at 23:28

@redmoskito: I have a question. How do you get the numeric value of the histogram of green for example ? So you can

subtract it with the other image histogram ? Let's say we have a green histogram with 3 pixel belonging to 0-63 bucket, and 5 pixel belonging to 64-127. Which is the value ?
– dynamic Jun 1, 2012 at 17:30

3 @Ikaso if its excactly the same image, you probably dont want to use anything like that and consider using simple CRC or MD5 comparison. If this is not sufficient, like there are single pixels that are different or metadata has changed, the histogram method is also sufficient. if your images are the same but rotated or scaled, a histogram based method can be sufficent but maybe will fail. if your images has changed colors you need to use interest point based algorithms.
– reox Jun 17, 2013 at 10:19

6 I'd like to add that nowadays, many fast alternatives to SIFT exist, such as the FAST detector and binary descriptors (BRIEF, BRISK, ORB, FREAK, BinBoost) to name a few. A tutorial on binary descriptors can be found here: gilscvblog.wordpress.com/2013/08/26/… – GilLevi Jun 19, 2014 at 14:24

101

The best method I know of is to use a Perceptual Hash. There appears to be a good open source implementation of such a hash available at:

http://phash.org/

The main idea is that each image is reduced down to a small hash code or 'fingerprint' by identifying salient features in the original image file and hashing a compact representation of those features (rather than hashing the image data directly). This means that the false positives rate is much reduced over a simplistic approach such as

reducing images down to a tiny thumbprint sized image and comparing thumbprints.

phash offers several types of hash and can be used for images, audio or video.

Share Improve this answer

Follow

answered Nov 9, 2011 at 22:17

redcalx

8,617 ● 8 ● 61 ● 108

Who's interesting in this method can find Objective-C Perceptual Hash realization by the link github.com/ameingast/cocoaimagehashing – Alexey Voitenko Jul 27, 2017 at 14:10

@AlexeyVoitenko Is this compatible with the hashes produced by phash.org in its default configuration? – Michael Apr 3, 2018 at 2:43

5 In my experience phash works well for finding different sizes of the same image, but not for similar images. Eg two different photos of the same object might have very different hashes. – Rena Jul 21, 2019 at 19:13

44

This post was the starting point of my solution, lots of good ideas here so I though I would share my results. The main insight is that I've found a way to get around the slowness of keypoint-based image matching by exploiting the speed of phash.

For the general solution, it's best to employ several strategies. Each algorithm is best suited for certain types

of image transformations and you can take advantage of that.

At the top, the fastest algorithms; at the bottom the slowest (though more accurate). You might skip the slow ones if a good match is found at the faster level.

- file-hash based (md5,sha1,etc) for exact duplicates
- perceptual hashing (phash) for rescaled images
- feature-based (SIFT) for modified images

I am having very good results with phash. The accuracy is good for rescaled images. It is not good for (perceptually) modified images (cropped, rotated, mirrored, etc). To deal with the hashing speed we must employ a disk cache/database to maintain the hashes for the haystack.

The really nice thing about phash is that once you build your hash database (which for me is about 1000 images/sec), the searches can be very, very fast, in particular when you can hold the entire hash database in memory. This is fairly practical since a hash is only 8 bytes.

For example, if you have 1 million images it would require an array of 1 million 64-bit hash values (8 MB). On some CPUs this fits in the L2/L3 cache! In practical usage I have seen a corei7 compare at over 1 Giga-hamm/sec, it is only a question of memory bandwidth to the CPU. A 1

Billion-image database is practical on a 64-bit CPU (8GB RAM needed) and searches will not exceed 1 second!

For modified/cropped images it would seem a transform-invariant feature/keypoint detector like SIFT is the way to go. SIFT will produce good keypoints that will detect crop/rotate/mirror etc. However the descriptor compare is very slow compared to hamming distance used by phash. This is a major limitation. There are a lot of compares to do, since there are maximum IxJxK descriptor compares to lookup one image (I=num haystack images, J=target keypoints per haystack image, K=target keypoints per needle image).

To get around the speed issue, I tried using phash around each found keypoint, using the feature size/radius to determine the sub-rectangle. The trick to making this work well, is to grow/shrink the radius to generate different sub-rect levels (on the needle image). Typically the first level (unscaled) will match however often it takes a few more. I'm not 100% sure why this works, but I can imagine it enables features that are too small for phash to work (phash scales images down to 32x32).

Another issue is that SIFT will not distribute the keypoints optimally. If there is a section of the image with a lot of edges the keypoints will cluster there and you won't get any in another area. I am using the GridAdaptedFeatureDetector in OpenCV to improve the distribution. Not sure what grid size is best, I am using a small grid (1x3 or 3x1 depending on image orientation).

You probably want to scale all the haystack images (and needle) to a smaller size prior to feature detection (I use 210px along maximum dimension). This will reduce noise in the image (always a problem for computer vision algorithms), also will focus detector on more prominent features.

For images of people, you might try face detection and use it to determine the image size to scale to and the grid size (for example largest face scaled to be 100px). The feature detector accounts for multiple scale levels (using pyramids) but there is a limitation to how many levels it will use (this is tunable of course).

The keypoint detector is probably working best when it returns less than the number of features you wanted. For example, if you ask for 400 and get 300 back, that's good. If you get 400 back every time, probably some good features had to be left out.

The needle image can have less keypoints than the haystack images and still get good results. Adding more doesn't necessarily get you huge gains, for example with J=400 and K=40 my hit rate is about 92%. With J=400 and K=400 the hit rate only goes up to 96%.

We can take advantage of the extreme speed of the hamming function to solve scaling, rotation, mirroring etc. A multiple-pass technique can be used. On each iteration, transform the sub-rectangles, re-hash, and run the search function again.

Share Improve this answer

Follow

answered Dec 1, 2013 at 20:21

wally
726 ● 7 ● 4

14

My company has about **24million** images come in from manufacturers every month. I was looking for a fast solution to ensure that the images we upload to our catalog are *new* images.

I want to say that I have searched the internet far and wide to attempt to find an ideal solution. I even developed my own edge detection algorithm.
I have evaluated speed and accuracy of multiple models. My images, which have white backgrounds, work extremely well with phashing. Like *redcalx* said, I recommend phash or ahash. **DO NOT** use MD5 Hashing or anyother cryptographic hashes. Unless, you want only EXACT image matches. Any resizing or manipulation that occurs between images will yield a different hash.

For phash/ahash, Check this out: imagehash

I wanted to extend *redcalx'*s post by posting my code and my accuracy.

What I do:

```
from PIL import Image
from PIL import ImageFilter
import imagehash

img1=Image.open(r"C:\yourlocation")
img2=Image.open(r"C:\yourlocation")
```

```
if img1.width<img2.width:
    img2=img2.resize((img1.width,img1.height))
else:
    img1=img1.resize((img2.width,img2.height))
img1=img1.filter(ImageFilter.BoxBlur(radius=3))
img2=img2.filter(ImageFilter.BoxBlur(radius=3))
phashvalue=imagehash.phash(img1)-
imagehash.phash(img2)
ahashvalue=imagehash.average_hash(img1)-
imagehash.average_hash(img2)
totalaccuracy=phashvalue+ahashvalue
```

Here are some of my results:

```
item1  item2  totalsimilarity
desk1  desk1       3
desk1  phone1     22
chair1 desk1      17
phone1 chair1     34
```

Hope this helps!

Share Improve this answer

Follow

edited Aug 31, 2019 at 14:39

answered May 2, 2019 at 20:04

Tanner Clark

691 ● 1 ● 8 ● 20

Is and MD5 has faster than an image hash? I'm just wondering about a potential situation where you have a fair number of identical images and if you would get a speed gain by doing an MD5 hash comparison first. – Pepsi-Joe Aug 30, 2022 at 10:41

8

As cartman pointed out, you can use any kind of hash value for finding exact duplicates.

One starting point for finding close images could be here. This is a tool used by CG companies to check if revamped images are still showing essentially the same scene.

Share Improve this answer

Follow

answered May 9, 2009 at 20:47

Tobiesque

762 ● 4 ● 7

7

I have an idea, which can work and it most likely to be very fast. You can sub-sample an image to say 80x60 resolution or comparable, and convert it to grey scale (after subsampling it will be faster). Process both images you want to compare. Then run normalised sum of squared differences between two images (the query image and each from the db), or even better Normalised Cross Correlation, which gives response closer to 1, if both images are similar. Then if images are similar you can proceed to more sophisticated techniques to verify that it is the same images. Obviously this algorithm is linear in terms of number of images in your database so even though it is going to be very fast up to 10000 images per second on the modern hardware. If you need invariance to rotation, then a dominant gradient can be computed for this small image, and then the whole coordinate system can be rotated to canonical

orientation, this though, will be slower. And no, there is no invariance to scale here.

If you want something more general or using big databases (million of images), then you need to look into image retrieval theory (loads of papers appeared in the last 5 years). There are some pointers in other answers. But It might be overkill, and the suggest histogram approach will do the job. Though I would think combination of many different fast approaches will be even better.

Share Improve this answer

Follow

answered Jun 11, 2009 at 21:22

Denis C

422 ● 5 ● 11

6

I believe that dropping the size of the image down to an almost icon size, say 48x48, then converting to greyscale, then taking the difference between pixels, or Delta, should work well. Because we're comparing the change in pixel color, rather than the actual pixel color, it won't matter if the image is slightly lighter or darker. Large changes will matter since pixels getting too light/dark will be lost. You can apply this across one row, or as many as you like to increase the accuracy. At most you'd have 47x47=2,209 subtractions to make in order to form a comparable Key.

Share Improve this answer

Follow

answered Nov 5, 2011 at 13:04

Tanoshimi

483 ● 1 ● 5 ● 11

4

What we loosely refer to as duplicates can be difficult for algorithms to discern. Your duplicates can be either:

1. Exact Duplicates
2. Near-exact Duplicates. (minor edits of image etc)
3. perceptual Duplicates (same content, but different view, camera etc)

No1 & 2 are easier to solve. No 3. is very subjective and still a research topic. I can offer a solution for No1 & 2. Both solutions use the excellent image hash- hashing library: https://github.com/JohannesBuchner/imagehash

1. Exact duplicates Exact duplicates can be found using a perceptual hashing measure. The phash library is quite good at this. I routinely use it to clean training data. Usage (from github site) is as simple as:

```
from PIL import Image
import imagehash

# image_fns : List of training image files
img_hashes = {}

for img_fn in sorted(image_fns):
    hash =
imagehash.average_hash(Image.open(image_fn))
    if hash in img_hashes:
        print( '{} duplicate of
{}'.format(image_fn, img_hashes[hash]) )
    else:
        img_hashes[hash] = image_fn
```

2. Near-Exact Duplicates In this case you will have to set a threshold and compare the hash values for their distance from each other. This has to be done by trial-and-error for your image content.

```
from PIL import Image
import imagehash

# image_fns : List of training image files
img_hashes = {}
epsilon = 50

for img_fn1, img_fn2 in zip(image_fns,
image_fns[::-1]):
    if image_fn1 == image_fn2:
        continue

    hash1 =
imagehash.average_hash(Image.open(image_fn1))
    hash2 =
imagehash.average_hash(Image.open(image_fn2))
    if hash1 - hash2 < epsilon:
        print( '{} is near duplicate of
{}'.format(image_fn1, image_fn2) )
```

Share Improve this answer

Follow

answered Apr 15, 2021 at 1:53

navneeth

1,317 ● 14 ● 21

Thanks. Could this be a good use case given below edaboard.com/threads/… Thanks & Regards,
– Prashant Akerkar May 25, 2021 at 11:01

3

Picking 100 random points could mean that similar (or occasionally even dissimilar) images would be marked as the same, which I assume is not what you want. MD5 hashes wouldn't work if the images were different formats (png, jpeg, etc), had different sizes, or had different metadata. Reducing all images to a smaller size is a good bet, doing a pixel-for- pixel comparison shouldn't take too long as long as you're using a good image library / fast language, and the size is small enough.

You could try making them tiny, then if they are the same perform another comparison on a larger size - could be a good combination of speed and accuracy...

Share Improve this answer

Follow

answered May 9, 2009 at 20:31

HarryM

1,885 ● 13 ● 7

1 If you're looking for exact duplicates but with different formats/metadata, you can do a hash (eg MD5) of the actual pixel values. Imagemagick calls this a signature (not related to cryptographic signing). You could also reduce it first, eg truncating to 4 bits per pixel to reduce impact of JPEG artifacts, or convert to grayscale to match slightly recolored images. – Rena Jul 21, 2019 at 19:16

2

If you have a large number of images, look into a Bloom filter, which uses multiple hashes for a probablistic but efficient result. If the number of images is not huge, then a cryptographic hash like md5 should be sufficient.

Share Improve this answer

Follow

answered May 9, 2009 at 20:26

jdigital

12.2k ● 4 ● 37 ● 54

So (trying to understand the Bloom filter) - does that mean you select random pixel points on the base image, randomly get either a red/green/blue value of the pixel - then compare to the new image? and then use a probability level (90% match) to determine how similar the two images are?
– meade May 9, 2009 at 20:57

6 This isn't a similarity check, it's an equivalence check. If you need similarity, then hashing is not the right approach. The idea behind Bloom is to use multiple hash algorithms to increase the likelihood of unique identification. Selecting random points isn't the best approach for a hashing algorithm because it will yield different results each time. – jdigital May 9, 2009 at 21:44

@jdigital The OP asked about being about to check if two images were "exact (or close)" - a hashing algorithm (such as MD5 or SHA1) would not be able to check if two images were close by not binary identical. – skeetastax Aug 22, 2023 at 8:06

1

I made a very simple solution in PHP for comparing images several years ago. It calculates a simple hash for each image, and then finds the difference. It works very nice for cropped or cropped with translation versions of the same image.

First I resize the image to a small size, like 24x24 or 36x36. Then I take each column of pixels and find average R,G,B values for this column.

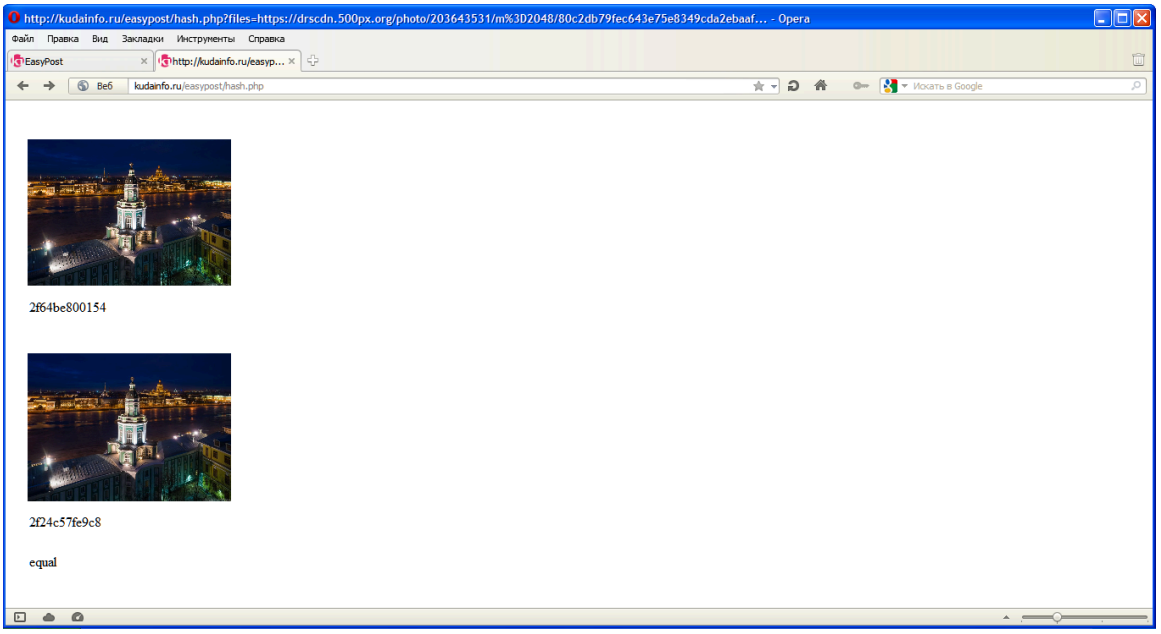
After each column has its own three numbers, I do two passes: first on odd columns and second on even ones. The first pass sums all the processed cols and then divides by their number (`[1] + [2] + [5] + [N-1] / (N/2)`). The second pass works in another manner: (`[3] - [4] + [6] - [8] ... / (N/2)`).

So now I have two numbers. As I found out experimenting, the first one is a major one: if it's far from the values of another image, they are not similar from the human point of view at all.

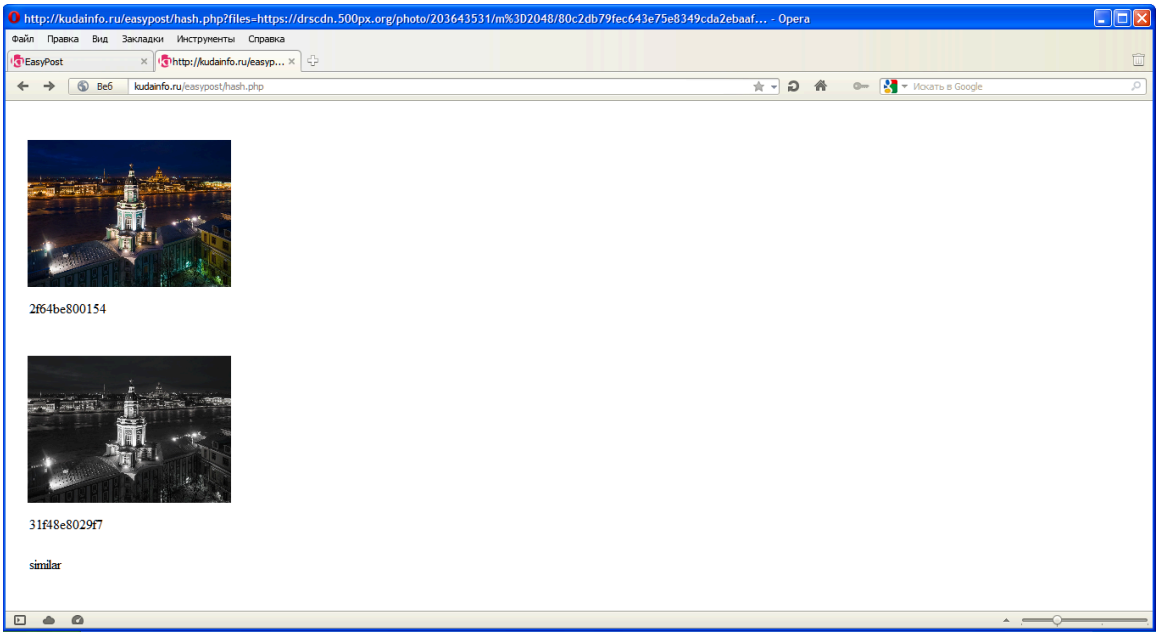
So, the first one represents *the average brightness* of the image (again, you can pay most attention to green channel, then the red one, etc, but the default R->G->B order works just fine). The second number can be compared if the first two are very close, and it in fact represents *the overall contrast* of the image: if we have some black/white pattern or any contrast scene (lighted buildings in the city at night, for example) and if we are lucky, we will get huge numbers here if out positive members of sum are mostly bright, and negative ones are mostly dark, or vice versa. As I want my values to be always positive, I divide by 2 and shift by 127 here.

I wrote the code in PHP in 2017, and seems I lost the code. But I still have the screenshots:

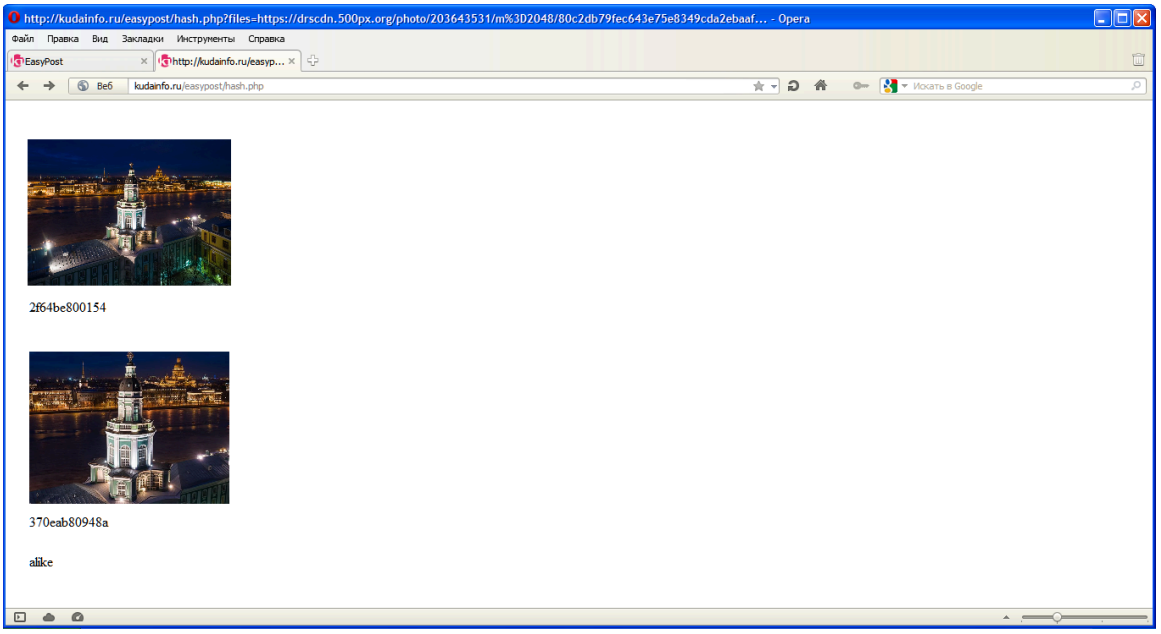
The same image:



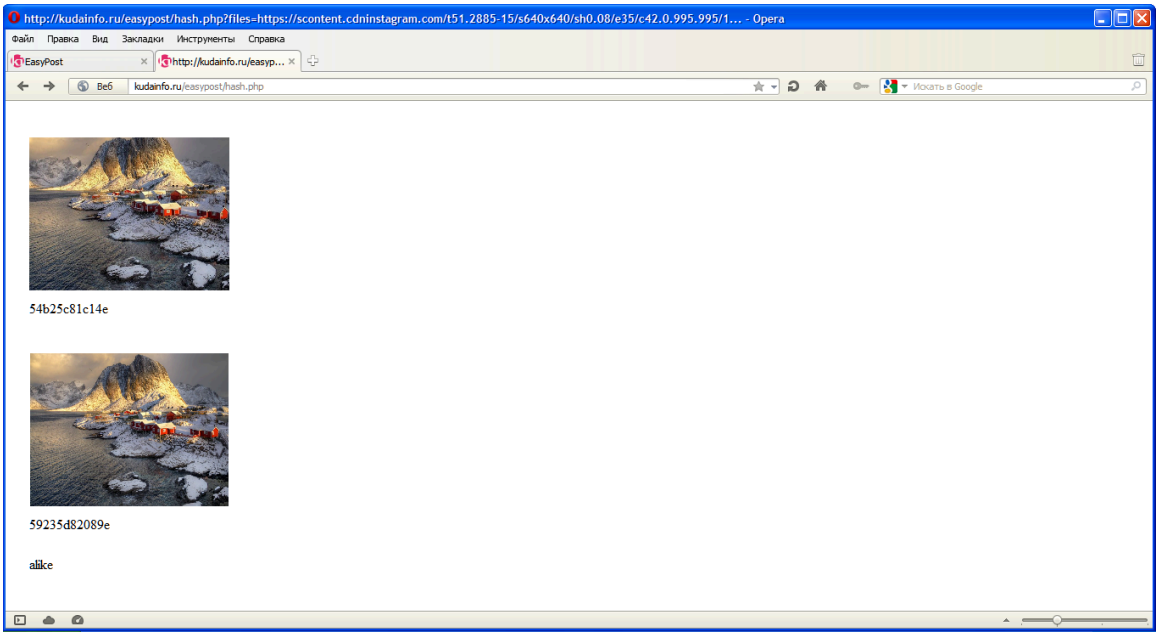
Black & White version:



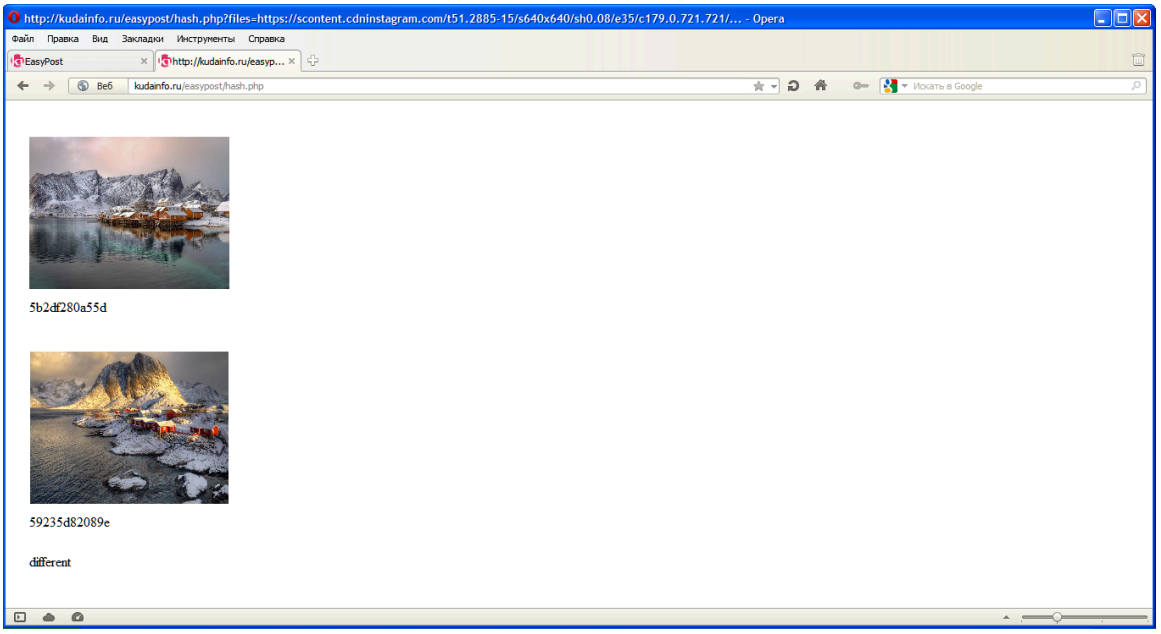
Cropped version:



Another image, ranslated version:



Same color gamut as 4th, but another scene:



I tuned the difference thresholds so that the results are really nice. But as you can see, this simple algorithm cannot do anything good with simple scene translations.

On a side note I can notice that a modification can be written to make cropped copies from each of two images at 75-80 percent, 4 at the corners or 8 at the corners and middles of the edges, and then by comparing the cropped variants with another whole image just the same way; and if one of them gets a significantly better similarity score, then use its value instead of the default one).

Share Improve this answer

Follow

answered Jan 14, 2023 at 4:54

Alex Popov

75 ● 1 ● 8

0

I think it's worth adding to this a phash solution I built that we've been using for a while now: Image::PHash. It is a Perl module, but the main parts are in C. It is several

times faster than phash.org and has a few extra features for DCT-based phashes.

We had dozens of millions of images already indexed on a MySQL database, so I wanted something fast and also a way to use MySQL indices (which don't work with hamming distance), which led me to use "reduced" hashes for direct matches, the module doc discusses this.

It's quite simple to use:

```
use Image::PHash;

my $iph1 = Image::PHash->new('file1.jpg');
my $p1   = $iph1->pHash();

my $iph2 = Image::PHash->new('file2.jpg');
my $p2   = $iph2->pHash();

my $diff = Image::PHash::diff($p1, $p2);
```

Share Improve this answer

Follow

answered Sep 26, 2022 at 12:40

Ecuador

1,167 ● 9 ● 28

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.