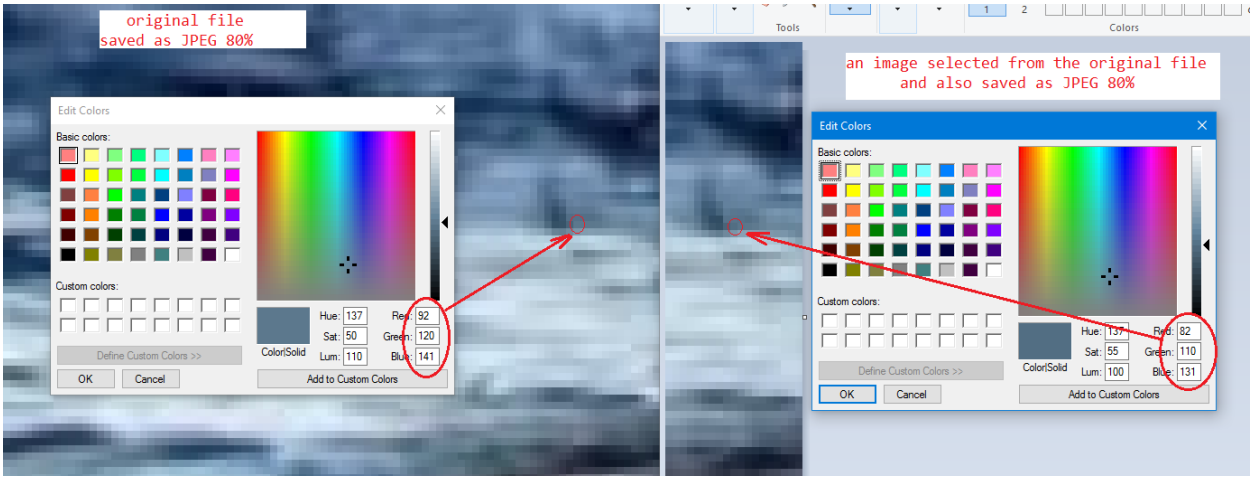
Overview

The original problem of *finding a location of cropped image inside an original one* is complicated by the fact that we have to deal with JPEG images. There is a good explanation of JPEG format and how the original image gets compressed: https://en.wikipedia.org/wiki/JPEG.

After studying the most interesting part - the steps involved into compression, it's easy to see that an image gets distorted when being saved as a JPEG file. The distortion depends in big part on the "quality" chosen. In fact, even if one chooses the **same** quality for the original and cropped images, the RGB colors (after loading and decompression) will be quite different in the original image region and the cropped version. Even more than that, different editors may compress images slightly different even if the same quality is chosen (due to potential 'downsampling' phase). And, last but not least, the color pallet of the image itself may affect the distortions during compression. Here is an example illustrating the point:



Without going into details, the searching algorithm should take into consideration the possibility of cropped image to be quite different in RGB color space.

Current solution

So, the current solution is straight forward and perhaps can be improved. It tries to find a location of sub-image, given some 'similarity' measure. The similarity here is defined as a maximum of Euclidian distances among all the RGB points of 2 images compared 1-to-1.

The algorithm basically:

- finds the biggest image first (which may be an original) just by looking at dimensions of images
- iterates over a bigger image, first checking for matching a 0-point (given the similarity threshold)
- if 0-point matches, then the full comparison is performed at that location

Testing

Besides a simple unittest, there is a full permutation test developed. I've picked some image from internet (of a 833 x 470 size), and first saved it as BMP image (to have a reference point), then selected 2 regions from it (also saved as BMP) and then slightly distorted 1 of those samples in 2 different ways (also saved as BMP).

So I had:

- *img/img-01/img-01.bmp* (original image)
- *img/img-01/img-01-smpl-01.bmp* (sample 1)
- *img/img-01/img-01-smpl-02.bmp* (sample 2)
- *img/img-01/img-01-smpl-04-n.bmp* (distorted sample)
- *img/img-01/img-01-smpl-03-n.bmp* (distorted sample)

Those images I've converted into JPEGs with different compression values. If you look into *img/img-01* folder, you'll see the file names follow the naming convention: ORIGINAL_BMP_NAME-COMPRESSION.jpg.

After that, the permutation test run, which tried to find the sampled images in the original ones. The results can be found in:

- *results/TestImg01Permutations-similarity-0.95.html* - tried to find subimages with 0.95% similarity
- *results/TestImg01Permutations-similarity-0.90.html* - tried to find subimages with 0.90% similarity
- *results/TestImg01Permutations-similarity-0.85.html* - tried to find subimages with 0.85% similarity

Summary

As it is easy to expect, the lower you set the similarity parameter, the more you find higher-compressed sub-images. On the other hand, there is a drawback: at some point you may run into situation where a not actual sub-images gets found, since it's close enough in terms of the distance measure.

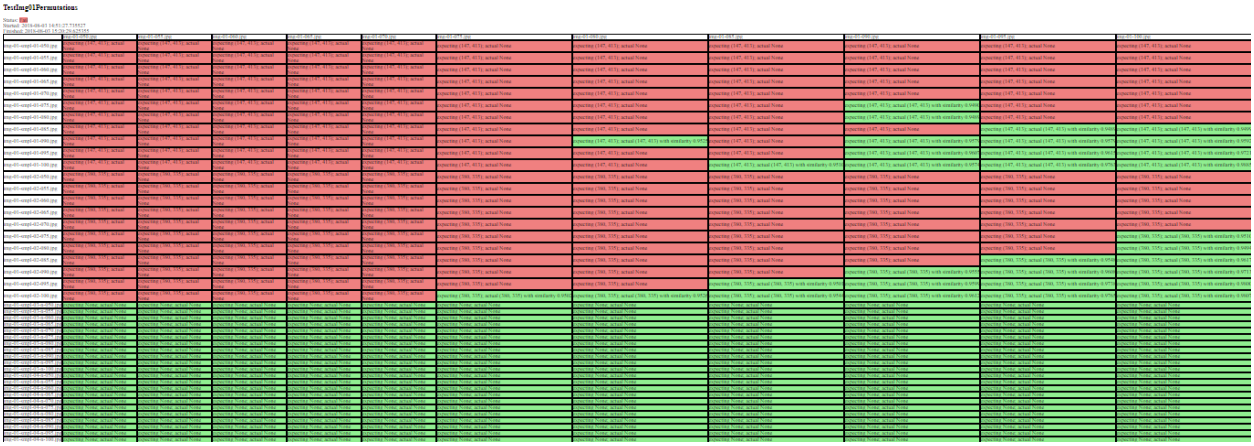
For example, for my worst-quality images I've seen:

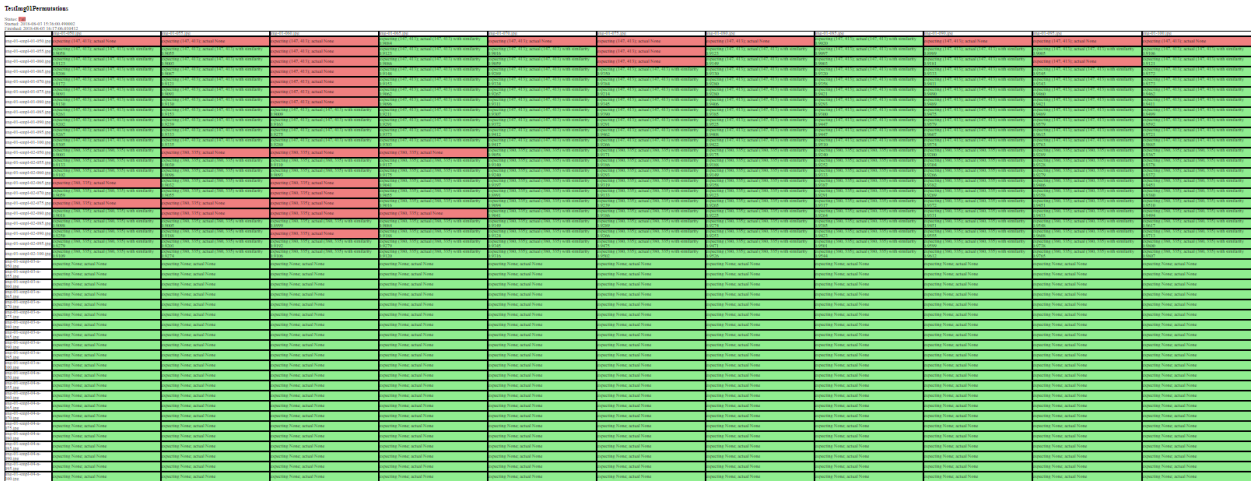
```
D:\Files\waldo-ag\subimage>python src/main/subimage.py img/img-01/img-01-050.jpg img/img-01/img-01-smpl-01-050.jpg -o
target/out.jpg -s .89
found 0.8914 similarity at (x,y) = (413,147)

D:\Files\waldo-ag\subimage>python src/main/subimage.py img/img-01/img-01-050.jpg img/img-01/img-01-smpl-01-050.jpg -o
target/out.jpg -s .8
found 0.7985 similarity at (x,y) = (413,146)
```

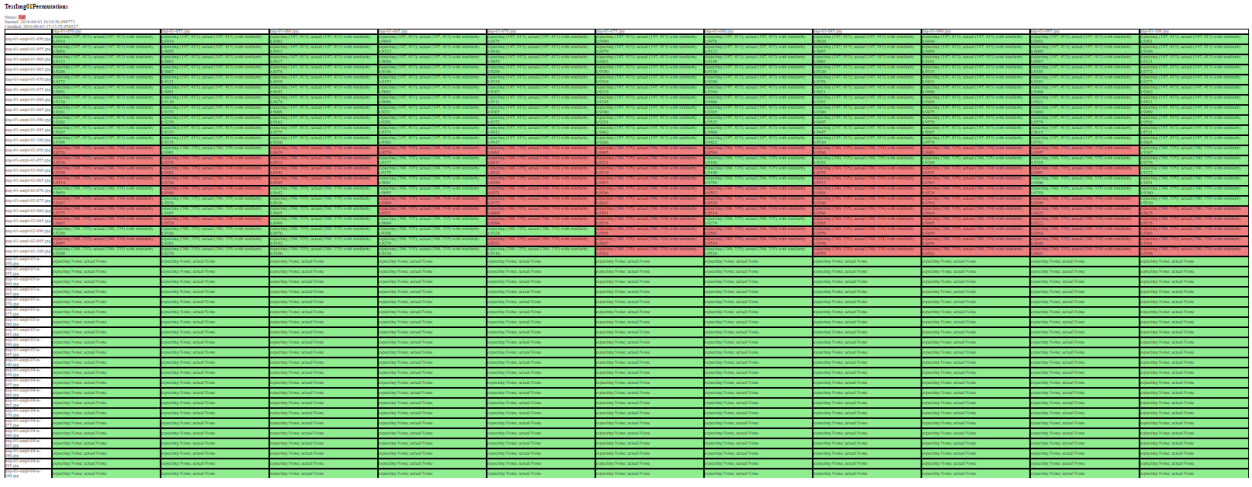
Notice how the Y coordinate differs and is slightly wrong at the second line. In this case that I had, the original image was compressed at JPEG 50%, and the sub-image was also at JPEG-50%. In order to find the sub-image I had to set the similarity measure to be 0.85.

In a normal case (when images are compressed at about 95%), a default similarity of 0.95 is sufficient enough.

Here is the conceptual look at permutation runs with different similarity measures:







As you can see, the matching becomes unstable once you start lowering the similarity measure.

Potential for improvements

The current runtime order is about **O**($(n-m)^2 * m^2$) where 'n' refers to width or height of 1 "bigger" image, and "m" refers to width or height of the "smaller" image. This is, obviously, too long. Unfortunately, the time constraints don't allow further investigation of this problem. There are multiple things can be tried such as:

- different pre-processing schema of images
- organizing image info into tree/hash-like structure (very similar to how JPEG actually gets compressed with "Discrete cosine transform")
- perhaps may try to use ML (like a NN) to improve a similarity measure of comparing 2 equi-sized JPEGs (but I'm not sure there will be any improvement in performance).
- etc