A solution to the Instagram engineering challenge

by Francesc Sala

Summary

The Instagram engineering challenge is published in:

https://engineering.instagram.com/instagram-engineering-challenge-the-unshredder-7ef3f7323ab1

A pdf copy of this challenge might be found also in file “challenge description.pdf”.

My solution is written in Python (I used Python 2.7.13 with PIL 1.1.7) and it consists of:

- One shredder script: shr.py , which is based on the one included in the challenge description. My shredder ensures that all shreds will have exactly the same width.
- Three versions of an un-shredder script: unshr_1.py, unshr_2.py, unshr_3.py. The three of them solve both parts of the challenge. The differences between them impact on performance and in the images they can un-shred. The main idea of my solution and the differences between the three versions will be detailed later on in this document.

Usage of the shredder: python shr.py *<input_image_filename> <num_of_shreds> <output_image_filename>*

Usage of the un-shredder: python unshr_*n*.py *<input_image_filename> <width_of_shreds>*
Or
python unshr_*n*.py *<input_image_filename>*

Depending on the parameters indicated to the un-shredder, it makes only the first part of the challenge or both. It always loads the input image, estimate the width of the shreds if not given as an input parameter, un-shreds the image, saves the un-shredded image to the file ‘output.png’ on the current working folder, and displays one miniature of the shredded image and one of the un-shredded image.

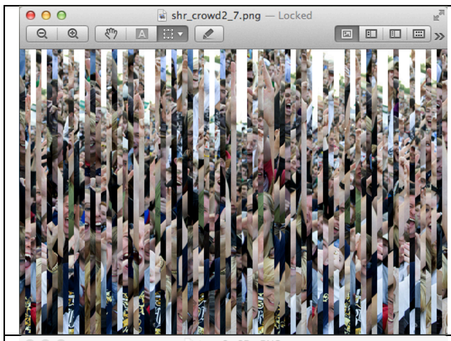
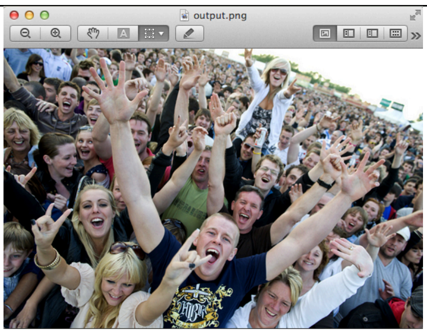
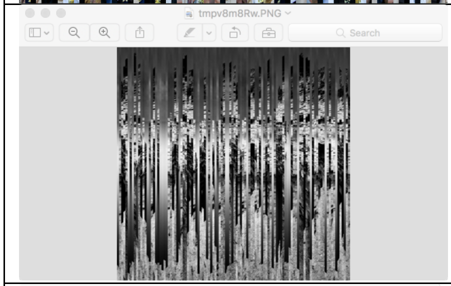
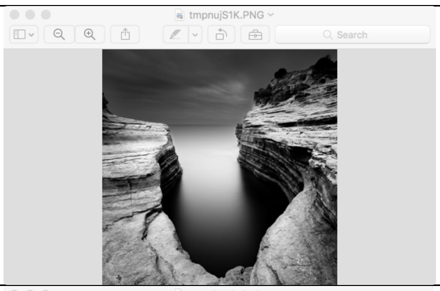
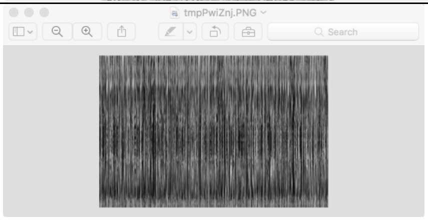
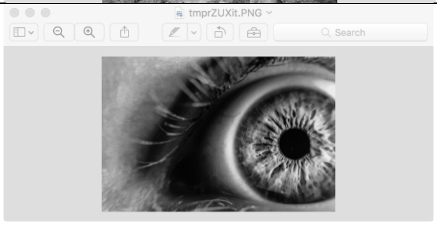
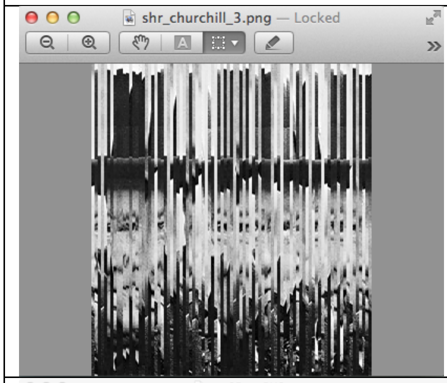
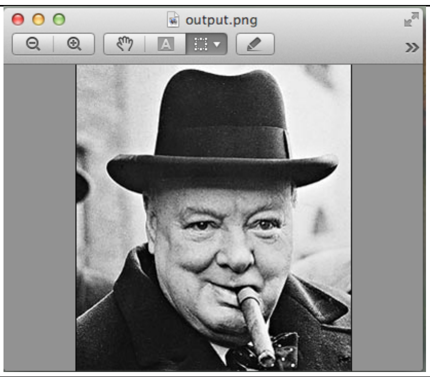
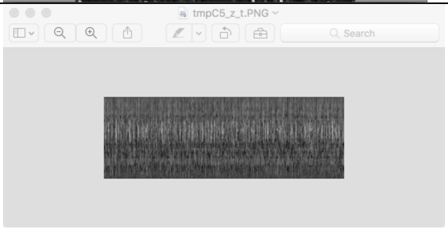
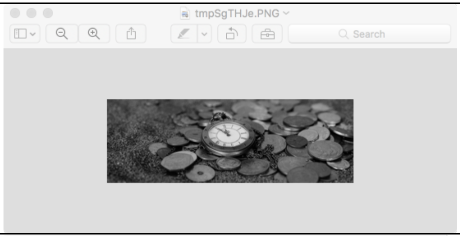
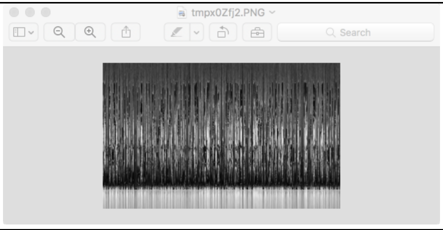
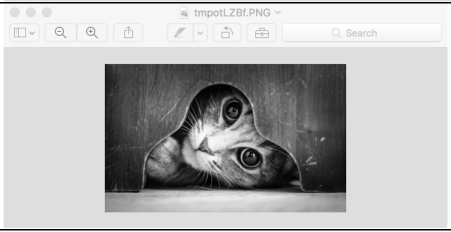
I tested with a test set of 45 color and grey shredded images (including the Tokyo one), with different sizes and different shred widths. All the images can be found in the folder ‘shredded_images’.

The three versions of the un-shredder are quite robust, and work correctly with the majority of the test images. They are able to work with shreds of any size, including shreds 3 pixels wide.

I tried with three images shredded in two-pixels wide shreds: one of them could be reconstructed correctly, but not the other two. I believe that when dealing with such narrow shreds, the algorithm can be confused with shreds belonging to a uniform background.

My three versions of the un-shredder estimate the width of the shreds (part two of the challenge) exactly in the same way. When doing this I assume that the image has at least 6 or 7 shreds. It does not work correctly if the images have fewer shreds.

On this page, some of the results with my un-shredders. The rest of this document is a detailed explanation of the solution.

	← 79 shreds of 7 pixels each is un-shredded to →	
	← 110 shreds of 5 pixels each is un-shredded to →	
	← 869 shreds of 4 pixels each is un-shredded to →	
	← 89 shreds of 3 pixels each is un-shredded to →	
	← 1280 shreds of 3 pixels each is un-shredded to →	
	← 440 shreds of 2 pixels each is un-shredded to →	

Part 1 of the challenge – reconstruction of image knowing width of shreds

Reconstruction strategy

Whatever strategy used to un-shred an image, it will be needed a reliable way to determine how good or bad any two shreds match (when placed one next to the other). I will explain later on the metrics I use to determine if two shreds match well or not. By now, just let's assume that we have such a reliable metric.

Let's number the shreds of the un-shredded image. An example below:

Image 1

distance(i , j) is such a numerical metric that measures how well shred j matches at the right of shred i. The smaller this distance, the better the two shreds match one next to the other

Using this distance, we can reliable determine, for any given shred i, which other shred j best matches at its right, and which other shred k best matches at its left.

So, using this distance it is possible to determine that $k - i - j$ go contiguous and in this order in the un-shredded image.

My reconstruction strategy is the same one in the three versions of the un-shredder. It is based on trying to identify as soon as possible the shred that is the leftmost one in the un-shredded image, and reconstructing from there.

The strategy assumes that the metric I have is completely reliable in identifying which shred goes next to another. The strategy is a combination of three steps.

Step 1: Try to identify the leftmost shred in the reconstructed image.

I construct vector bestNeighboursRight[*0* . . *S*-*1*], where:

- *S* is the number of shreds
- For any shred *i*, bestNeighboursRight[*i*] is the shred *j* (*j* ≠ *i*) that minimizes distance(*i* , *j*). In other words, it is the shred that better matches *i* from the right; the shred that goes to the right of *i* in the reconstructed (un-shredded) image.

If everything goes as expected the vector bestNeigboursRight has no repetitions, except perhaps because of the rightmost shred in the un-shredded image.
For the rightmost shred of the un-shredded image it does not make sense to calculate the shred that best matches it to the right. It will give one, any, with no special meaning. But I do not know in advance which is that rightmost shred, so I calculate for all of the shreds. Likely, when calculating for the rightmost shred in the un-shredded image it will introduce a repetition in the vector.

Let's see an example, for an image with 12 shreds:

bestNeighboursRight

0	1	2	3	4	5	6	7	8	9	10	11
10	4	11	2	7	9	8	6	5	4	1	0

It has exactly one repetition, 4 (marked in red). And from that I know:

a. That the rightmost shred in the un-shredded image is either shred 1 or shred 9 (marked in green, the two indexes where the repetition occurs).
b. That the leftmost shred in the un-shredded image is the missing value in the vector: shred 3 in the example.

Once identified the leftmost shred, and having computed the vector bestNeighboursRight, it is straightforward to reconstruct the image, from left to right.

Step 2: Insist on identifying the leftmost shred in the reconstructed image.

Suppose that step 1 did not allow identifying the leftmost shred. For example:

bestNeighboursRight

0	1	2	3	4	5	6	7	8	9	10	11
10	4	11	2	7	9	8	6	5	3	1	0

There are no repetitions in the vector, and then there are no missing values either. Then it is not possible to identify leftmost or rightmost shreds.
This happens when the shreds that best matches the rightmost one on its right is the leftmost shred. It is as if the sorted sequence of shreds formed a cycle, like a cylinder.

What I do then is to compute an analogous vector: bestNeighboursLeft[*0* . . *S-1*], where:

- *S* is the number of shreds
- For any shred *i*, bestNeighboursLeft[*i*] is the shred *j (j ≠ i)* that minimizes distance(*j*, *i*). In other words, it is the shred that better matches *i* from the left; the shred that goes to the left of *i* in the reconstructed (un-shredded) image.

For any "interior" shred of the image, and because the vectors were constructed based on reliable metrics, one can expect that:

if bestNeighbourRight[*i*] = *j* then bestNeighbourLeft[*j*] = *i*

If I am able to find a shred *j* where the previous condition does not hold, that one must be the leftmost shred.

If identified the leftmost shred, the image can be easily reconstructed.

Step 3. Reconstruct the image from an arbitrary shred

Still it can happen that steps one and two did not allow the identification of the leftmost shred.

bestNeighboursRight

0	1	2	3	4	5	6	7	8	9	10	11
10	4	11	2	7	9	8	6	5	3	1	0

bestNeighboursLeft

0	1	2	3	4	5	6	7	8	9	10	11
11	10	3	9	1	8	7	4	6	5	0	2

A scenario like this one means that the leftmost and rightmost shred in the un-shredded image match each other very strongly.

As I do not know which is the leftmost shred, I cannot reconstruct from left to right.
As I do not know which is the rightmost shred, I cannot reconstruct from right to left either.

The strategy then is as follows:

Start with any shred, *0* for instance. The vectors bestNeighboursLeft and bestNeighboursRight gives two other shreds that best matches the shred *0* for the left and for the right. Compute then distance(*0,* bestNeighboursRight[*0*]) and distance(bestNeighboursLeft[*0*], *0*). The smallest of these two values tells me which shred add to the solution, and if adding to the left or to the right.

Suppose I have added shred *11* to the right: my reconstruction up to now is [*0*, *11*]

Now, I go on computing distance(*11*, bestNeighboursRight[*11*]) and distance(bestNeighboursLeft[*0*], *0*). The smallest of these two values tells me which shred add to the solution, and if adding to the left or to the right.

And it is done like this at every step, adding to the left or adding to the right the shred with smaller distance, until a sequence of *S* shreds is obtained.

Metric to measure how well two shreds match when placed one next to the other

Putting together (side by side) any two shreds of the image, if they are contiguous in the un-shredded image, the image they form will be continuous and smooth.
But if the shreds are not contiguous, the image they form will not be smooth at all; there will be a vertical edge along the shred boundaries.

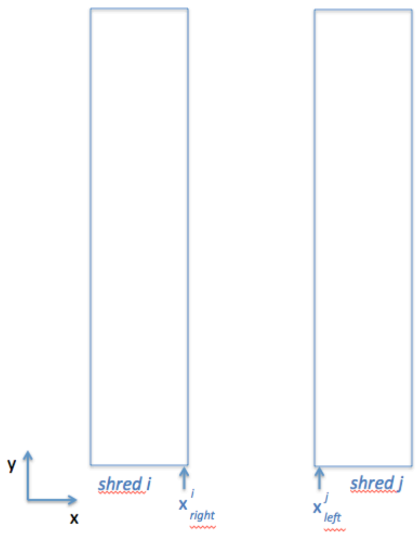


Image 2

I have used two different metrics:

Metric 1: it is just taking the difference of pixel values along the adjacent shred edges.

Given two any shreds, *i* and *j*,

$$distance(i,j) = \sum_{y=1}^{N} |f\left(x^{i}_{r}, y\right) - f\left(x^{j}_{l}, y\right)|$$

where

N is the height of the image

x_l^j is the leftmost coordinate of shred j

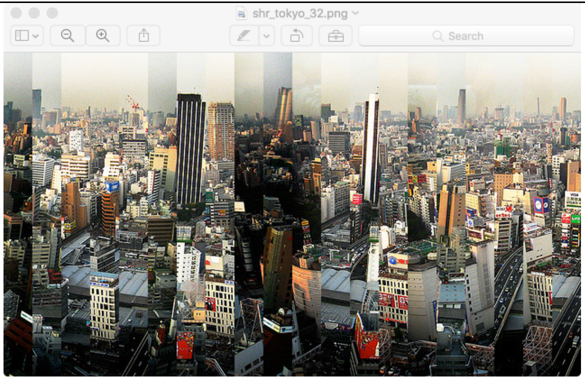
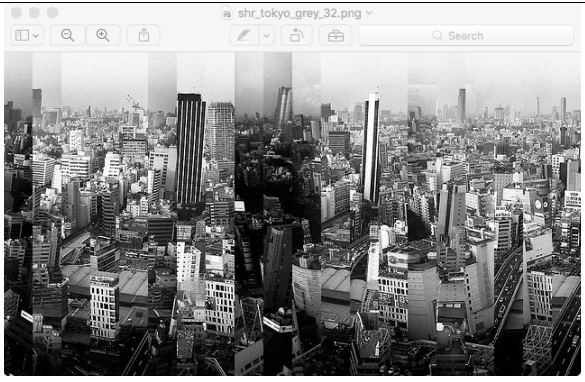
x_r^i is the rightmost coordinate of shred i

$f(x, y)$ is the value of pixel at coordinates x, y

This distance tells how well the shred *i* fits when placed to the left of shred *j*. The smaller the distance, the better the fit.

$f(x, y)$ can either be a vector (color images, or grey images with identical R,G,B channels) or can be a scalar (grey images with just one channel). It is a matter just to apply a norm if dealing with vectors.

This metrics works very well with almost of the images of the test set, including the sample picture of Tokyo in the Instagram challenge description. But it does not work well if that same sample were grey instead of color:

 shr_tokyo_32.png	Is un-shredded correctly when using this metric.
 shr_tokyo_grey_32.png	Is **not** un-shredded correctly when using this metric.

Metric 2: It is a pity failing on the sample image in grey, so I try a second metric:

$$distance(i,j) = \sum_{y=1}^{N} \left| f\left(x_{l}^{j} + 1, y\right) - 3\, f\left(x_{l}^{j} + 1, y\right) + 3\, f\left(x_{r}^{i}, y\right) - f\left(x_{r}^{i} - 1\right) \right|$$

which corresponds to adding up the differences of lateral second derivatives of f

As the explanation on how to arrive to the expression above might result a little bit tedious, I have placed in a separate file: "solution annex I – Second Derivatives".

With this second metric, both Tokyo pictures (color and grey) are correctly un-shredded. However, metric 2 fails in image where metric 1 does not. So, both metrics are quite good, but none of them is perfect.

Part 2 of the challenge – calculation of the width of the shreds

Well, it is an estimate rather than a calculation, because what I do is to apply a couple of heuristics that let me guess a width.

First of all, calculate for every column of pixels in the shredded image, its distance to the next column, using the function *distance* already explained before. Then order the indexes of the columns according the distances obtained in each one, from bigger to smaller. So, at the end of this step I have a sorted list of indexes, each one representing a column in the image. The indexes on the top of the list are the ones with biggest *distance* function, so the ones that presented a sharp discontinuity, and at the bottom of the list, those that have a smaller *distance*, where image was smooth. Let's call *L* to this sorted list.

Then I have to consider every submultiple of the width of the image, as a potential width of the shreds. For each one of these candidates I will apply a heuristic measure, and will take that one with best heuristic measure.

Let's suppose I am considering *w* as a potential width of the shreds in the shredded image, and that with that width there would be *S* shreds in the image (width of the image is *w * S)*.
The heuristics I apply are two:

- The borders of the shreds would be the columns numbered as *w, 2w, 3w,* … and so on. If really the image has been shredded in shreds of *w* pixels, one could expect that those indexes occupy the first *S* positions in the list *L.*

  I look for the position of those columns w, 2w, 3w, … in *L*, and add those positions.
  If they were occupying the first *S* positions, the calculated addition would be exactly this amount.

  $$\sum_{i=1}^{S} i = \frac{s * (s+1)}{2}$$

  So I measure how close the calculated addition is to the formula above, by making a division of both quantities. The closest it is to 1, the most likely is that *w* is the width we are looking for.

- The heuristic above has a problem. Let's suppose *w* is the width of the shreds. It can happen that when considering *2w* as a width, it takes better heuristic measure, and that would lead me to a wrong identification of the width, which would not allow reconstructing the image properly.

  I need then a second heuristic, or somehow to make a correction to the first one. What I do is to take again the addition of the positions in *L* of columns *w, 2w, 3w, …* and divide that addition by the number of shreds, *S.*

Finally, what I do is to take that width *w* that minimizes the multiplication of both heuristics, which means that is a good candidate according to the two heuristics.

What I just explained above is not working very well when there are really few shreds, so I am not considering all the possible widths of shreds in the algorithm. Those that would result in very few shreds I just ignore. Ignoring them is not a problem:

- If the shredded image has not that few shreds I am not making any mistake by ignore those candidate widths.
- If the shredded image has few shreds, what I will be probably doing is guessing a submultiple of the real shred width, so breaking down every shreds in several ones during the reconstruction algorithm. But at the end the reconstruction will give the same result. (If the width is 20, it is a problem to predict 40, but it is not to predict 10. With 40 one cannot reconstruct the image successfully, but can with 10).

Test

Image test set

I took eighteen images and shredded them (with sript shr.py) using different widths, to obtain a test set of forty-seven shredded images, including the one of the city of Tokyo used in the blog of Instagram to illustrate the challenge.
Original (un-shredded) images can be found in folder "original_image", while shredded images can be found in folder "shredded_images".
I used, for the shredded images, a naming convention that helps me to have at hand the width of the shreds when testing the un-shredders.
Every shredded image has a name of the format:

shr_<name_of_original_image>_<width_of_shreds>.png

For instance:
shr_rose_32.png is a shred of image rose.jpg made with shreds of 32 pixels.
shr_churchill_3.png is a shred of image churchill.jpg made with shreds of 3 pixels.

Test Results

As commented, I have written three un-shredder scripts, but the three of them use the same reconstruction strategy and the three of them estimate the widths of the shreds in the same way. So, they differ very little: they differ in the distance function they use, and in the pixels they work with.

Script	Pixels	Distance function (metric)
unshr_1.py	Works with all the channels of the image.	Difference of pixel values (n channels) along shred borders. Uses a loop.
unshr_2.py	Works with a single channel	Difference of pixel values (one channel only) along shred borders. Array version: does not use a loop.
unshr_3.py		Difference of second derivatives along shred borders. Array version: does not use a loop.

Script unshr_1.py is robust, but a little bit slow.
By working only with one channel and making an array-based version of the metric function, scripts unshr_2.py and unshr_3.py are much faster.
Detailed results of the three scripts on the test set images can be found in file: "solution annex II - Results Tables.pdf"