Final Project System Implementation: PatchMatch

R08922079 洪浩翔 R08944056 黃子晟 R08944034 洪商荃

About this project

In this project, we implement Patch Matching, the work of Barnes et al., with Python as our final project. The main usage of our program includes two parts. The first one is reconstruction. It can transfer patches from source image to target image. That is, the same as original work did, using patches from source image to reshuffle the target image. The second part is hole filling with easy constraint. Our program can fill a hole on an image with only one constraint. For details, our implementation includes three main parts in original paper, including initialization, propagation, and searching, and one part of hole filling tool.

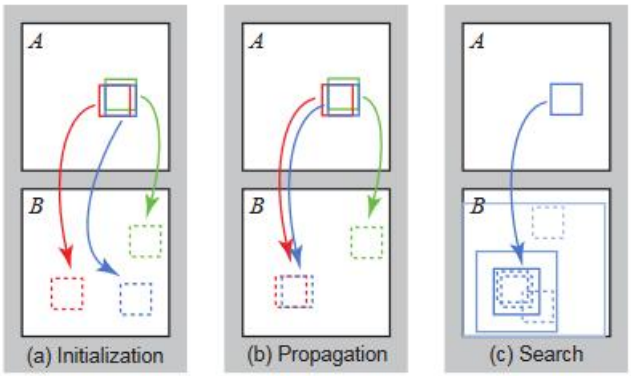

Figure (1). Executing pip-line of original paper. After initialization, process (b) and (c) are executed iteratively until the program stops.

Initialization

Instead of using prior information, we initialize our program with random reference. We create some position pairs in the range of height and width of input image randomly as our initial reference.

Distance

To calculate distance of two patches, we simply follow the method of L2 norm. When getting two different patches, subtracting them followed with squaring and summing the values would get the final result. Return the resultant value as the distance of two patches.

Propagation

We follow the details of propagation mentioned in paper to implement this part. In odd iterations, we compare current reference of position (x,y) with reference in position (x-1,y) and (x,y-1), and use the best result, which means smallest distance, to update value in reference map to the best reference. In even iterations, we compare the reference of current position with those in position (x+1,y) and (x,y+1), and do the same thing as in odd iterations. After all positions are calculated, the process moves to the next step - searching.

Searching

Differing from original paper which starts from the input image size as the searching area, we use small initial searching area in the beginning to speed up the process. The rest is the same as original paper. We decrease searching area with multiplying searching height and width with a constant 0.5, and stop the process when searching radius is below or equal to 1. When getting a better distance value at certain patch in the searching area, update the reference map to the better position and repeat the process again until convergence. If the process still not achieves the final iteration, it goes back to propagation and repeats these two parts again.

Hole filling tool

Differing from the paper, we design a naïve hole filling algorithm by ourselves. First, we take hole and constraint masks as additional inputs, and separate hole mask to two sub-masks based on the given constraint. If there is no constraint, the hole mask is used directly. Second, we set the hole part to NaN based on input sub-masks respectively in order to avoid effects on calculating distance. Last, the program iterates through the whole image with a given stride, and if it find a pixel in hole part, it would crop a small patch and find a best match through patchMatch method. However, if using crop-and-patch method, the result would have apparent artifacts, especially when the stride is large. As a result, we average pixel values if the patching place is not NaN. Otherwise, we patch the best match directly.

Experiment

We set up several experiments to examine our implementation.

- Reconstruction

(a) Reference | (b) Target | (c) Reconstruction result

Figure (2). Reconstruction. Reference image is where the patches come from, and target image is reconstructed target. Use reference image to reconstruct target image. In the final result, reconstructed image still preserves its similarity but has a style-transfer like artifacts.

➢ Hole filling w/o constraint

(a) Input (b) Hole (c) Filling result

Figure (3). Hole filling w/o constraint. We mask input images based on corresponding hole masks and recover them. For there is no constraint in this experiment, we mainly mask places that have low frequency texture. One can see that our naïve algorithm can generate well-looking recovery result on these textures. However, the algorithm has limited ability on recovering meticulous details.

- Hole filling with constraint 1

Input

Hole

Constraint

Masked part 1

Masked part 2

Filling result

Figure (4). Hole filling with constraint. We separate hole mask based on constraint mask, and filling two holes respectively to preserve edge. Due to the measurement error when creating constraint, the recovered line is not as expected. It is not a straight line. Despite that, color of recovered patch still match the original image.

- Hole filling with constraint 2

Input

Hole

Constraint

Filling result

Figure (5). Hole filling with constraint. The constraint here is much precise, so it can preserve the edge better. However, as describe in Figure (3), meticulous details are hard to recover.

Usage

To sum up, our implementation can be used not only in reconstruction and common patch matching to produce a style-transferred-like results, but also can be used in hole filling task. Although we don't implement some complex extended tools based on our implementation, it is also easy to use our code as the backbone of such tools. For different platform, we remain simple API of our patchMatch code with parser so that others can use it easily by importing it or calling in command line.

Reference

C. Barnes, E. Shechtman, A. Finkelstein, and D. Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. ACM Transactions on Graphics, 2009.