Exploration of Patch-Based Image Processing Methods

Tony Korol
tkorol@ucsb.edu
UCSB ECE 178
Santa Barbara, California, USA

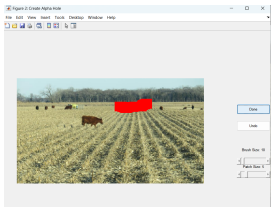

Figure 1: GUI Interface

Figure 2: Patch Fill Results

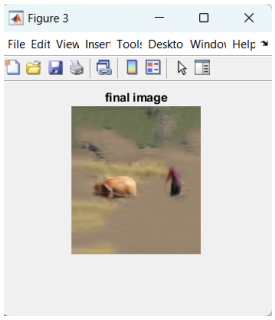

Figure 3: Image Retargeting Results

ABSTRACT

I propose a comprehensive and intuitive process to implement two important patch-based image processing algorithms: Patch-Based-Synthesis *Barnes et. al. 2009* and Patch-Based-Image-Retargeting *Simakov et. al. 2008*. This article explains a logical approach to implementing hole-filling and image size reduction, starting with a slow, brute force patch search implementation, and gradually optimizing performance to allow for good results within one sitting. Furthermore, this article details the creation of an easy to use Graphical Interface for users to interactively select a hole region to fill in any image.

ACM Reference Format:
Tony Korol. 2024. Exploration of Patch-Based Image Processing Methods. In *Proceedings of 3/22/2024 (UCSB ECE178).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

If you have ever used Photoshop before, you may recognize the term "Content Aware Fill". This functionality, which I will call hole-fill from now on, is a way to remove sections of an image while still maintaining a coherent scene. Image retargeting, on the other hand is the process of changing the size of an image while maintaining the maximum level of coherence *and* completeness in the image. In order to implement both of these measures, the method of patch search and vote needs to be introduced.

All patch search and vote does is find the closest matching patches in a Source image and map those patches to a target image. So if we have a source image with certain details and we want those details to be reflected in the target image, we can use patch search and vote to apply source patches to the target. This process requires several steps. First, we must create a Nearest Neighbor Field (NNF), which is a coordinate mapping of each target pixel to the closest corresponding patch in the source image. This requires a us to calculate the distance between each patch, which will be discussed in more detail. The next step is to populate the final target image with the source patches specified by the NNF and weigh them appropriately to obtain the final color of each pixel in the target image. This process can be done in several ways and I will discuss some optimizations to it in this article.

Now that we understand the general idea of patch search and vote, we can discuss implementation. The naive way to implement such a task is by iterating through every target pixel and for each target pixel, iterating through all source patches to find the one with minimal distance from the target patch. This results in $O(N^2 * M^2 * PatchSize^2)$ time complexity for a target of NxN dimensions and source of MxM dimensions. This is a huge waste of time, which is why we implement the PatchMatch algorithm detailed in the Barnes Article. The PatchMatch algorithm exploits the coherent nature of images, as well as the optimal behavior of random search algorithms. The steps of this algorithm are as follows: Random Initialization, where we choose an initial random NNF for all target pixels, Propagation, in which we propagate good matches from neighboring target pixels to surrounding pixels, and finally, random search, which randomly updates each pixel between iterations in order to avoid reaching local minima in our optimization.

We implement hole fill and image retargeting by cleverly assigning the target and source in each scenario to best fit the desired end result. There are several ways to make the PatchMatch algorithm even even more effective at solving these problems, like using multiscale search and enhanced distance metrics to find better matches.

UCSB ECE178, Santa Barbara, CA,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

These algorithms will be discussed in more detail in sections 3 and 4.

2 PATCH MATCH IMPLEMENTATION

2.1 Brute Force Implementation

For the brute force implementation of patch search and vote, I came across several issues and solutions, most of which still did not result in optimal time complexity.

Firstly, I had to find a way to ensure that there were valid values available for every target patch and source patch in the images, and that accessing pixels on the edge of the images wouldn't cause an error in operation. The way I solved this is by padding the source image with mirror padding, which allows for more usable fill content that otherwise would not be available on the edge of the image. For the target image, I padded it with NaN values, because we do not want to introduce extra structure to the target image which does not exist. By using NaN padding, I was then able to use logical masking in later steps to determine patch distance only for non-NaN values.

The rest of the code went as follows: Iterate through every pixel in both images and find the best match for each target pixel in the source image. One huge problem I had with this code is that comparing the distance for one patch over and over many times resulted in huge increases in run time. After submitting part 1 of my project, I was able to slim down my distance implementation (L2 norm) into two lines using Matlab vector builtin functions, which reduced the runtime by a factor of 2.

The final thing left to complete in this case was the VoteNNF function, which takes all of the patches specified in the NNF and votes them into one final target patch. I did this by "splatting" all of the patches on top of each other around their respective coordinates, and subsequently dividing each pixel by the amount of patches that contributed. To figure out how many patches contributed at each point, I noticed that for each index within half of a patch of the edge of the image, we lose a contribution factor of $patchSize * (patchSize - 1)/2 - index))$ where $patchSize$ is the side length of the patch and index is the index of the pixel along the dimension perpendicular to the edge. Thus, once all of the patches are splatted, we can do an element-wise division by the contribution factor we have calculated for each pixel and this gives us our final image.

2.2 PatchMatch Implementation

Because of the tedious and time-complex nature of the brute force algorithm for patch matching, we implemented PatchMatch as described in Barnes et. al. This method results in a time complexity on the order of $O(iterations * TargetWidth * TargetHeight * patchSize^2)$ which is a factor of N^2 better than the brute force algorithm. This allowed my code to go from taking upwards of 200 seconds to compute an NNF to a mere 0.63 seconds with Patch Match.

The most challenging part of this algorithm was actually understanding what was going on under the hood. I first implemented it under the assumption that the program first iterates through all target pixels and carries out propagation, and then does a whole separate set of iterations to carry out random search, however, this obviously ended up creating a random NNF, which sometimes even exceeded indexing bounds of the target and source images. Thus I implemented forced boundaries on NNF values in order to ensure every NNF maps to a valid source patch. I also had to revise my entire implementation to perform both the propagation step and the random search step within each iteration.

Figure 4: PatchMatch Source Image

Figure 5: PatchMatch Target Image

Figure 6: PatchMatch Result Image

In general, after fixing this misconception, the program was able to accurately perform propagation and search for each target pixel from left top to right bottom in even iterations, which would be propagated down to the bottom right in odd iterations when the order of iteration reverses. I was able to get the algorithm to converge after 5 iterations, which shows the power of random search in finding global minima for our optimization.

3 HOLE FILL IMPLEMENTATION

3.1 Graphic User Interface

The way hole fill is used in practice is by allowing the user to select a region in an image which they would like to remove, and subsequently fill that region with the most coherent information available from the rest of the image. I created a Graphical User Interface (GUI) for users to load in matlab, which allows the user to select brush size, patch size, and draw on the image where they would like to make the hole. I used documentation from the matlab GUI page to implement this functionality.

When the user runs the program "contentAwareFill.m", they are presented with a pop up to choose the image which they wish to operate on. Once the image is chosen from the user's file system, the user can then use the sliders on the side of the screen to choose the brush width to draw on the image. to draw, the user must left-click and drag along the image. When the user is done drawing, the algorithm should automatically begin to fill the hole and should result in "output.m" being returned.

3.2 Hole Fill Algorithm

My hole fill algorithm had many steps that needed to be fulfilled in order to make sure that the correct patches were being used and no

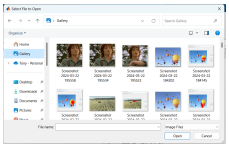

Figure 7: GUI File Selection

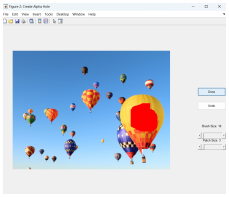

Figure 8: GUI Hole Painting

invalid information was applied to the target image. The first order of business is to identify the hole region and find its dimension. From the GUI, we already get an image with an Alpha Channel hole, which leaves us with the problem of finding the largest side dimension of the hole. This, I did by assigning a binary mask for the Alpha hole region and iteratively scaling inward toward the hole area until the bounds of the iteration reach the edge of the hole region. This process can be seen in holeMaxDim.m. The next step is to resize the image by numScales down to a size at which the hole region is on the order of the patch size. When I did this, I also had to resize the alpha channel in order to later use it to create the smaller alpha masks.

Once you have your stack of resized images, you must take the smallest one and initialize the hole region so that all target patches used are valid. This can be done using matlab interpimg on each of the RGB channels.

From here, you must be very careful to create what I call a "target mask" which informs the NNF search algorithm which target pixels to update and which ones to leave at 0 offset. The pixels that need to be updated are the ones in the hole region itself as well as ones within half of a patch distance of the hole region. The way I obtained this region is by doing a convolution of the alpha mask with an impulse function of a patch full of ones, which conveniently created a buffer region of valid target pixels to iterate over. I also inverted this mask to create a source pixel mask which informs the NNF algorithm which pixels are valid mappings to choose from when doing random initialization and propagation.

In the PatchMatchNNF algorithm itself, I had to update several things. Firstly, I set most of the NNF to be 0, except for the area in the target mask. Then, I added several conditional statements to ensure that the offsets applied to these target pixels would be within the valid source region.

Finally, this algorithm wraps up with the VoteNNF function just like the previous patch match algorithm. This is the easy part, because now that we have the NNF there is no more complicated masking logic that we need to implement. At this stage, I noticed that using my previous implementation of the patch vote algorithm, I was getting little to no detail in my filled holes, which was not optimal when trying to conserve image coherence. To combat this issue, I decided to implement a weighting scheme based on patch

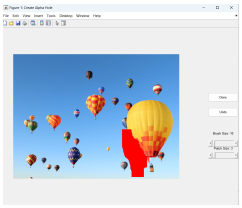

Figure 9: Paint for Background Fill

Figure 10: Background Fill

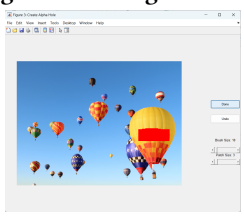

Figure 11: Paint for Detail Fill

Figure 12: Detail Fill

distance. In this scheme, source patches mapped to the target pixel which have a higher L2 norm distance from the target patch are weighed less than those with more optimal distance. This allowed the patch vote algorithm to preserve more detail because the combined patches would not have to be naively averaged at the end of the patch vote process.

I have found that using higher patch sizes allows for higher detail, but also makes the process of calculating the NNF and voting much more time consuming, so I have generally stuck to a patch size of 3 pixels which gives a decent estimate of the image in the hole region.

4 PATCH BASED RETARGETING

4.1 General Algorithmic Process

To implement Patch Based Retargeting, I first set up the main project file in which I set up the coarse scale image, performed the retargeting at this coarse scale until reaching the desired aspect ratio, and then finally scaled up to the desired target size.

Figure 13: Retargeting Source

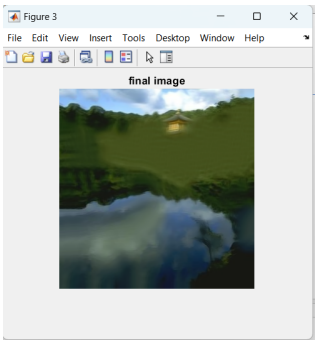

Figure 14: BDS Result

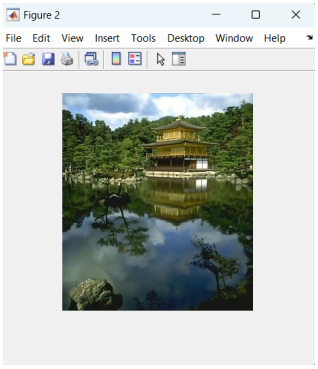

Figure 15: Seam Carve Result

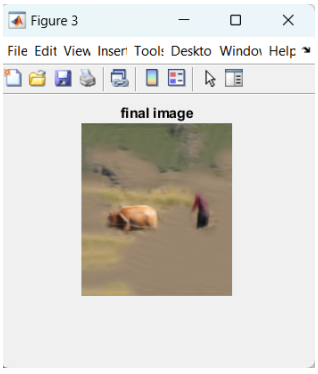

Figure 16: BDS Result Simakov Farmer

In each of these steps the main objective is to be able to iteratively change the dimensions of the target image in small steps while running a patch search and vote process in each iteration to conserve image fidelity.

Ideally this would be done by running a NNF search on the target to the image and again vice versa to obtain a Bidirectional mapping of source and target pixels. This allows you to calculate a bidirectional similarity metric (BDS) which measures both coherence (accuracy of the target with respect to the source) and completeness (presence of all source patches in the target). By measuring the NNF in both directions, when we implement the vote function, we can weigh both target and source colors to determine the final contribution to the image. By voting high BDS patches in with higher weights, we can guarantee that at each iteration, we are maximizing the coherence and completeness of the image.

4.2 Abridged Implementation

In my attempt to create the BDS voting algorithm, I ended up not very successful, with my program taking an excessive amount of time to run, and outputting a majority black pixels. This is likely due to the weighting scheme of my BDS vote algorithm because it seems like when I do the normalization after "splatting" the necessary patches down, I lost all of the detail of the image.

Because of these issues, I decided that I would just implement the general high level algorithm, only with the coherence term as we had been doing earlier. For the first step of downscaling and iterating until reaching the required aspect ratio, I got decent results, likely due to the small amount of change between each iteration. The issue came when there were large steps in upscaling more heavily to get to the final target dimensions. In my code when I run the basic patch search and vote, it ends up causing a lot of loss of detail because of the inability to tune bidirectional similarity. The general color gradients are preserved, however any high detail parts that would ideally be preserved are not prioritized by BDS and hence lost.

For future implementations, I am considering reading much more deeply into the Simakov paper in order to correctly and fully implement BDS search and vote and I would also like to extend the algorithm to work on upscaled images.

4.3 Comparison to Seam Carving

In my case, because the algorithm is clearly not ready to implement bidirectional similarity, I am not really able to make an honest comparison to the seam carving algorithm. A few things I noted however are that seam carving is a much faster and more scalable algorithm, while in an ideal implementation of the BDS algorithm, the BDS algorithm would beat seam carving in terms of result fidelity.