भारतीय प्रौद्योगिकी संस्थान खड़गपुर
Indian Institute of Technology Kharagpur

DIGITAL IMAGE PROCESSING GROUP PROJECT

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

DEPARTMENT OF ELECTRONICS & ELECTRICAL COMMUNICATION ENGINEERING

Region Filling and Object Removal by Exemplar-Based Image Inpainting

Team Members:
Anand Jhunjhunwala - 17EC35032

Arpit Dwivedi - 17EC35005

Pankaj Mishra - 17EC35034

Parakh Agarwal - 17EC35016

Guide:
Prof. Saumik Bhattacharya

October 24, 2020

Contents

Part I

Introduction

In this assignment we implement the inpainting algorithm presented by Criminsi et al. in 2004 for removing large objects from digital photographs and replacing them with visually plausible backgrounds. Before this paper was published, the problem was addressed by two classes of algorithms:

1. Texture synthesis algorithms for generating large image regions from sample textures. These algorithms performed well in tasks related to replicating consistent textures but faced difficulty in filling large holes in real-world images.This is because such images contain multiple textures interacting spatially.

2. Inpainting techniques for filling in small image gaps. These algorithms work by propagating linear structures (isophotes) into the target region through diffusion. However this technique introduced significant blur when filling large regions.

The paper presented a novel and efficient algorithm that had the efficiency and qualitative performance of exemplar-based texture synthesis algorithms and also respected the image constraints imposed by surrounding linear structures.

Features of proposed Algorithm

1. **Exemplar based synthesis approach**
   The algorithm used an isophote-driven image sampling process to fill the target region. Exemplar-based texture synthesis is sufficient for propagating extended linear image structures. All we need to do is to find the best match source patch. The isophote orientation is preserved in this manner and both texture and structure information is propagated. This patch based approach is used because it is significantly faster as compared to the pixel-based approach. The region to be filled, i.e., the target region is indicated by Ω, and its contour is denoted by $\delta\Omega$. The source region, $\Phi = \mathcal{I} - \Omega$, which remains fixed throughout the algorithm, provides samples used in the filling process. Suppose that the square template $\Psi_p \in \Omega$ centred at the point p (as shown in 1), is to be filled. The best-match sample from the source region comes from the patch $\Psi_{\hat{q}} \in \Phi$, which is most similar to those parts that are already filled in Ψ_p. In the figure we can see that if Ψ_p lies on the continuation of an image edge, the most likely best matches will lie along the same (or a similarly coloured) edge (e.g., $\Psi_{q'}$ and $\Psi_{q''}$ in 1). We observe that the isophote orientation is automatically preserved.

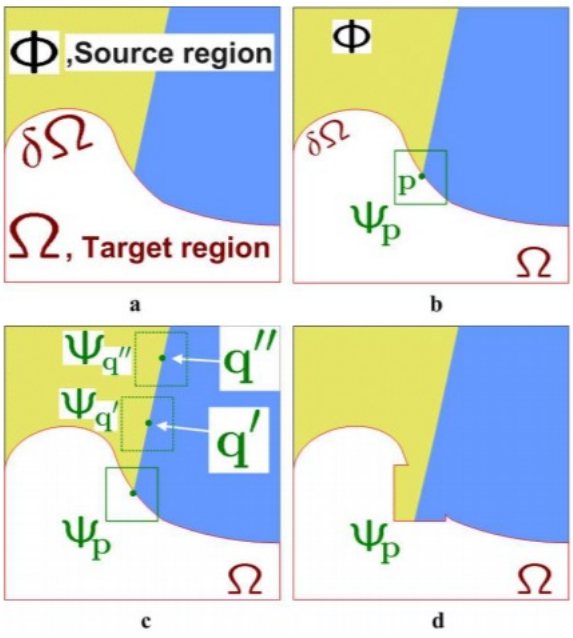

Figure 1: Structure propagation by exemplar-based texture synthesis.

2. **Appropriate filling order**
   The concentric layer filling approach used before this paper was published was referred to as the onion peel method. However this method didn't produce the expected results as it didn't prioritise the regions to be filled. Ideally higher priority should be given to those parts which lie on the continuation of image structures. The proposed algorithm achieves a good balance between the propagation of textured regions and structured regions. It avoids "over-shooting" artefacts that occur when image edges are allowed to grow indefinitely. The comparison between the two methods is shown in 2

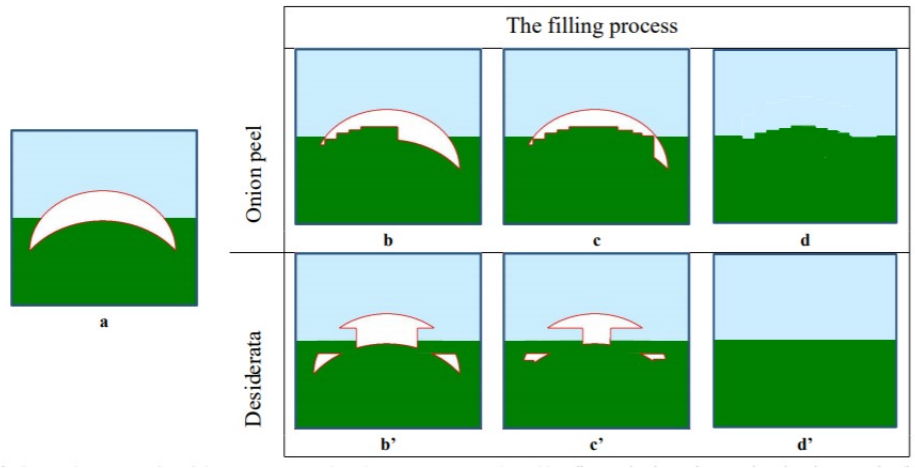

Figure 2: Importance of the filling order when dealing with concave target regions.

Algorithm Details

First, given an input image, the user selects a target region, Ω, to be removed and filled. The source region, Φ, may be defined as the entire image minus the target region. Then the size of the target window is specified whose default value is 9X9. Each pixel maintains a colour value (empty if the pixel is unfilled) and a confidence value, which reflects our confidence in the pixel value, and which is frozen once a pixel has been filled. During the course of the algorithm, patches along the fill front are also given a temporary priority value, which determines the order in which they are filled.

1. The priority P(p) is computed by multiplying the confidence C(p) and data D(p) terms. The confidence term is higher for those pixels that have a large number of pixels in their neighbourhood already filled (or belonging to the source region). This enforces the concentric filling order as boundary pixels have higher confidence values as compared to the center pixels. The data term is of fundamental importance in the algorithm because it encourages linear structures to be synthesized first, and, therefore propagated securely into the target region. This helps to achieve the desired structure propagation.

$$\begin{aligned} \text{P(p)} &= \text{C(p)*D(p)} \\ \text{C(P)} &= \frac{\sum_{q \in \Psi_p \cap (\mathcal{I}-\Omega)} C(q)}{|\Psi_p|} \\ \text{D(p)} &= \frac{|\nabla I_p^{\perp} . n_p|}{\alpha} \end{aligned}$$

   where $|\ \Psi_p\ |$ is the area of Ψ_p, α is a normalization factor (e.g., $\alpha = 255$ for a typical grey-level image), n_p is a unit vector orthogonal to the front $\delta\Omega$ in the point p and $\perp$ denotes the orthogonal operator. The priority P(p) is computed for every border patch, with distinct patches for each pixel on the boundary of the target region.

2. After the patch with the highest priority is found, it is filled with the data extracted from the source region. Earlier techniques involved using diffusion but that resulted in significant blur especially for large regions and so a different technique was used. The image texture

was propagated into the source region and the most similar patch was searched. The value of each pixel-to-be-filled in the target patch was then copied from its corresponding position inside the source patch.

$$\Psi_{\hat{q}} = \mathrm{argmin}_{\Psi_q \in \Phi}\, \mathrm{d}(\Psi_{\hat{p}}, \Psi_q)$$

where the distance $\mathrm{d}(\Psi_a, \Psi_b)$ between two generic patches Ψ_a and Ψ_b is simply defined as the sum of squared differences of the already filled pixels in the two patches. Pixel colours are represented in the CIE Lab colour space because of its property of perceptual uniformity.

3. After the patch $\Psi_{\hat{p}}$ has been filled with new pixel values, the confidence C(p) is updated in the area delimited by $\Psi_{\hat{p}}$ as follows

$$\mathrm{C(p)} = \mathrm{C}(\hat{p})\ \forall p \in \Psi_{\hat{p}} \cap \Omega$$

Final Algorithm

Algorithm 1: Final Algorithm

Input: $\mathcal{I}$: Colour Image to perform Inpainting.
Ω: User selected area in Image to fill using Image Inpainting.

Algo: Extract the manually selected initial front $\delta\Omega^0$
Repeat until done:
1a. Identify the fill front $\delta\Omega^t$. If $\Omega^t = \oslash$, exit.
1b. Compute priorities $P(p)\ \forall\ p \in \Omega^t$.
2a. Find the path $\Psi_{\hat{p}}$ with the maximum priority, *i.e.* $\hat{p} = \mathrm{argmax}_{p \in \delta\Omega^t} P(p)$.
2b. Find the exemplar $\Psi_{\hat{q}} \in \Phi$ that minimizes $\mathrm{d}(\Psi_{\hat{p}}, \Psi_{\hat{q}})$.
2c. Copy image data from $\Psi_{\hat{q}}$ to $\Psi_{\hat{p}}\ \forall\ p \in \Psi_{\hat{p}} \cap \Omega$.
3. Update C(p) $\forall\ p \in \Psi_{\hat{p}} \cap \Omega$.

Output: Final region filled Image.

Part II

Implementation

This section of report explains the overall code flow, and functions that we used to Implement the above explained paper.

Functions Definition

The paper has been implemented in the form of various independent functions that each serve a specific purpose, along with a main function which integrates everything into a single program. The primary functions we wrote for the program are:

1. **Patch_dim**

```
function [px,py] = patch_dim(x,y,H,W,dfx,dfy)
```

Input: Image dimensions H*W, default patch size dfx*dfy, and pixel coordinate (x,y)
Role: Since it is not always possible to have a patch of dimension 9x9 around every contour point (for instance, when the point is very close to an image boundary), we make use of this function to find the largest rectangular patch (of length and breadth less than or equal to 9 pixels each) that can be formed around the given pixel (center of patch).

2. **findPriority**

```
function [pix_x, pix_y, conf\_level] = findPriority(b,mask,conf_mat, d_mat)
```

Input: Boundary points bp, binary mask of image, confidence matrix, data vector of bp points.
Role: Given a list of contour points bp along with their data terms and the confidence matrix, it computes the priority of all the contour points and returns the coordinate of the point with maximum priority along with the maximum confidence value.

3. **findSimilarPatch**

```
function [q_x,q_y] = findSimilarPatch(px, py, bx, by, img, mask)
```

Input: Patch dimension px*py, patch center (bx,by), image in CIE Lab format, binary mask
Role: Once the pixel with highest priority has been determined along with the patch which surrounds it, this function scans the entire region searching for patches which lie completely in the known region and have minimum euclidean distance from the given input patch and returns its center along with the maximum confidence value.

4. **findDistance**

```
function [d] = findDistance(pixA, pixB)
```

Input: Two pixel coordinates:- PixA and PixB.
Role: This is an auxiliary function that assists the findSimilarPatch function. Given 2 pixels in CIE Lab format, it returns the euclidean distance between them.

5. **LineNormal2D**

```
function [N] = LineNormal2D(bp)
```

Input: List of contour points bp.
Role: In addition to the above functions, we also use this open-sourced function written by D.Kroon, to compute the normal vector at each contour point. Due attribution has been given in the code comments.

Code Flow

This section describes the overall code flow and function calls.
The sequence of steps is as follows:

1. **Clear the environment variables and read the image file:**

```
clc;
clear;
close all;
imtool close all;
workspace;

fontSize = 8;
imgfile = "xyz.jpg";
dpx = 9;
dpy = 9;

Image = imread(imgfile);
imshow(Image);
axis on;
title('Original Image', 'FontSize', fontSize);
```

   In the above snippet, imgfile hold image name, dpx and dpy are the default patch dimensions and imread is used to read the provided image.

2. **Take the input mask from user:**

```
set(gcf, 'Position', get(0,'Screensize'));
message = sprintf('Select the region you want to remove from image.\n\nTo
    select: Left click and hold to begin drawing.\nSimply lift the mouse
    button to finish.\n\nPress OK to start');
uiwait(msgbox(message));
hFH = imfreehand();
mask = hFH.createMask();

Image = double(Image);
imgsize = size(Image);
bsize = size(mask);
mask_removed_image = Image.*repmat(~mask, [1,1,3]);
imshow(mask_removed_image/255);
```

   In the above snippet, 'imfreehand' object is used to select the mask region followed by .createMask() method. The mask_removed image is shown using imshow and this image gets updated after every iteration.

3. **Compute gradient of image after removing masked region:**

```
[Ix(:,:,3), Iy(:,:,3)] = gradient(mask_removed_image(:,:,3));
[Ix(:,:,2), Iy(:,:,2)] = gradient(mask_removed_image(:,:,2));
[Ix(:,:,1), Iy(:,:,1)] = gradient(mask_removed_image(:,:,1));
Ix = sum(Ix,3)/(3*255); Iy = sum(Iy,3)/(3*255);
temp = Ix; Ix = -Iy; Iy = temp;
```

   In the above snippet, 'gradient' method is used to compute the gradient at each pixel of the masked removed image where each channel's gradient is computed individually, and then averaged over all the channels.

4. **Find and boundary points of current mask:**

```
boundary = bwboundaries(mask);
bp = boundary{1};
for i=2:size(boundary,1)
    b = boundary{i};
    bp =  vertcat(bp, b);
end
```

In the above snippet,**bwboundaries** function is used to compute the boundary points of the current mask and this is also referred to as a contour in the paper.

5. **Compute data terms of all boundary points:**

```
Ipx = zeros(size(bp,1), 1);
Ipy = zeros(size(bp,1), 1);
for k = 1:size(bp, 1)
    Ipx(k) = Ix(int32(bp(k,1)),int32(bp(k,2)));
    Ipy(k) = Iy(int32(bp(k,1)),int32(bp(k,2)));
end
N = LineNormals2D(bp);
N(~isfinite(N))=0;

D = double(abs(Ipx.*N(:,1) + Ipy.*N(:,2)))/255;
```

In the above snippet, Data terms of all boundary points are computed using **LineNormals2D** function. Image gradient is also calculated at the boundary points using the normalized absolute dot product between the two.

6. **Find boundary point with maximum priority:**

```
lab = rgb2lab(mask_removed_image);
[bx, by, max_priority_C] = findPriority(bp, mask, C, D);
```

In the above snippet, the boundary point with the maximum priority is calculated using **findPriority** function.

7. **Find Similar patch in known region:**

```
[px, py] = patch_dim(bx, by, size(Image,1), size(Image,2), dpx, dpy);
[mx, my] = findSimilarPatch(px, py, bx, by, lab, ~mask);
```

In the above snippet, **findPriority** function is used to find the patch with the minimum distance, i.e maximum similarity with the above selected patch.

8. **Copy the pixels from most similar patch:**

```
%update_image
for x = ceil(-px/2):floor(px/2)
    for y = ceil(-py/2):floor(py/2)
        source_r = mask_removed_image(int32(mx+x), int32(my+y), 1);
        source_g = mask_removed_image(int32(mx+x), int32(my+y), 2);
        source_b = mask_removed_image(int32(mx+x), int32(my+y), 3);

        if C(int32(bx+x), int32(by+y)) cop u<= 0
          mask_removed_image(int32(bx+x), int32(by+y), 1) = source_r;
          mask_removed_image(int32(bx+x), int32(by+y), 2) = source_g;
          mask_removed_image(int32(bx+x), int32(by+y), 3) = source_b;

          Ix(int32(bx+x), int32(by+y)) = Ix(int32(mx+x), int32(my+y));
          Iy(int32(bx+x), int32(by+y)) = Iy(int32(mx+x), int32(my+y));
        end
    end
end
```

In the above snippet, unknown pixels of the selected patch are filled by copying the corresponding pixel's RGB values and image gradient from the most similar patch found above.

9. **Update the confidence term:**

```
%update_C matrix
for x = ceil(-px/2):floor(px/2)
    for y = ceil(-py/2):floor(py/2)
        if mask(int32(bx+x), int32(by+y)) == 1
          C(int32(bx+x), int32(by+y)) = max_priority_C;
        end
    end
end
```

In the above snippet, all the pixels with unknown confidence values are filled with the confidence value of the maximum priority patch. This is done using a nested loop, one pixel per iteration.

10. **Update the mask:**

```
%update_mask
for x = ceil(-px/2):floor(px/2)
    for y = ceil(-py/2):floor(py/2)
        mask(int32(bx+x), int32(by+y)) = 0.0;
    end
end
imshow(mask_removed_image/255);
fprintf("Number of pixels left in mask: %d\n", sum(mask(:)));
```

In the above snippet, all the pixels that were filled in the above iteration, set their mask coordinates to 0 to signify that they have been filled.

Part III

Results

This section of report includes our implementation results on different images.

Example1: In this example we tried to reproduce linear structure of image.

Figure 3: Original Image

Figure 4: Image after removing mask

Figure 5: Final Result

Example2: In this example we tried to remove the upper pillar from the image.

Figure 6: Original Image

Figure 7: Final Result

Example3: In this example we tried to remove the middle pillar from the sidewalk.

Figure 8: Original Image

Figure 9: Final Result

Example4: In this example we tried to remove leftmost human figure from the image.

Figure 10: Original Image

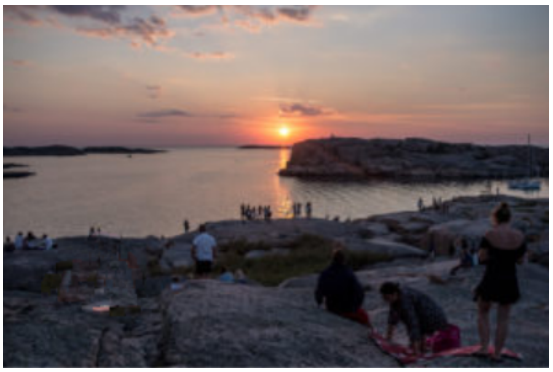

Figure 11: Final Result

Example5: In this example we tried to remove middle giraffe from the image.

Figure 12: Original Image

Figure 13: Final Result

Example6: In this example we tried to remove right side person from the image

Figure 14: Original Image

Figure 15: Image after removing mask

Figure 16: Final Result

Part IV

Conclusion

This section of report includes our observation and discussion of the above results.

- This algorithm performs very well as it propagates linear structures into the target region, as it can be seen in 5.
- Since this algorithm directly copies a patch from the image so we can see in 13 that a clear boundary is formed after inpainting. This is one of the drawbacks of this algorithm.
- In the case of a highly textured image, it is very difficult to identify the target region boundary as can be seen in 7.
- As can be seen in 16 one small unwanted patch (showing leg) had been copied. This is due to the fact that both the players' dress and the background are white in colour thereby creating a camouflage effect.
- The time complexity of this algorithm grows exponentially with the image size as for each patch it has to traverse the whole image to find a similar patch. This is also the slowest step of the algorithm.
  In our case it takes approximately 2 hours to fill 10,000 pixels when the input image has a size of 1000*1300.
- Since we have used nested loops for implementation of this algorithm, our code takes more time than specified in the paper. One solution to this problem is making use of vectorization to avoid nested loops thereby making the code run much faster.
- We observe this algorithm performs well irrespective of the shape of of the shape and topology of the target region. Moreover it preserves the sharpness of edges and implements balanced region filling to avoid over-shooting artefacts.
- However this algorithm does not produce great results if similar patches do not exist within the image.

Bibliography

Criminisi, A., Perez, P. & Toyama, K. (2004) Region filling and object removal by exemplar-based image inpainting. *IEEE Transactions on Image Processing*. 13 (9), 1200–1212.

MATLAB version 9.3.0.713579 (R2017b) (2017). The Mathworks, Inc. Natick, Massachusetts.