KAIST
EE535 Digital Image Processing
Assignment 3

Honi Selahaddin (20226089)

May, 2022

Inpainting

Image inpainting refers to filling the missing parts while preserving the contextual integrity in the image. In this report, it is aimed to analyze *Region filling and object removal by exemplar-based image inpainting* (Criminissi, 2004) algorithm and show the experimental results. Also, it is discussed how to improve prediction quality with implementation details on reference source code. [1]

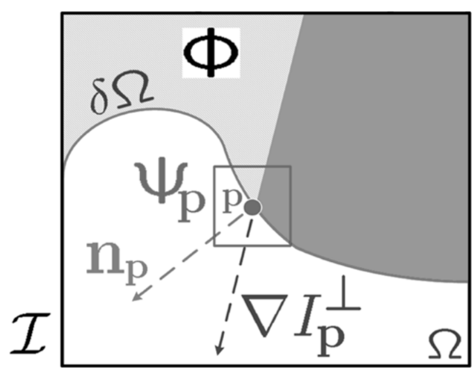
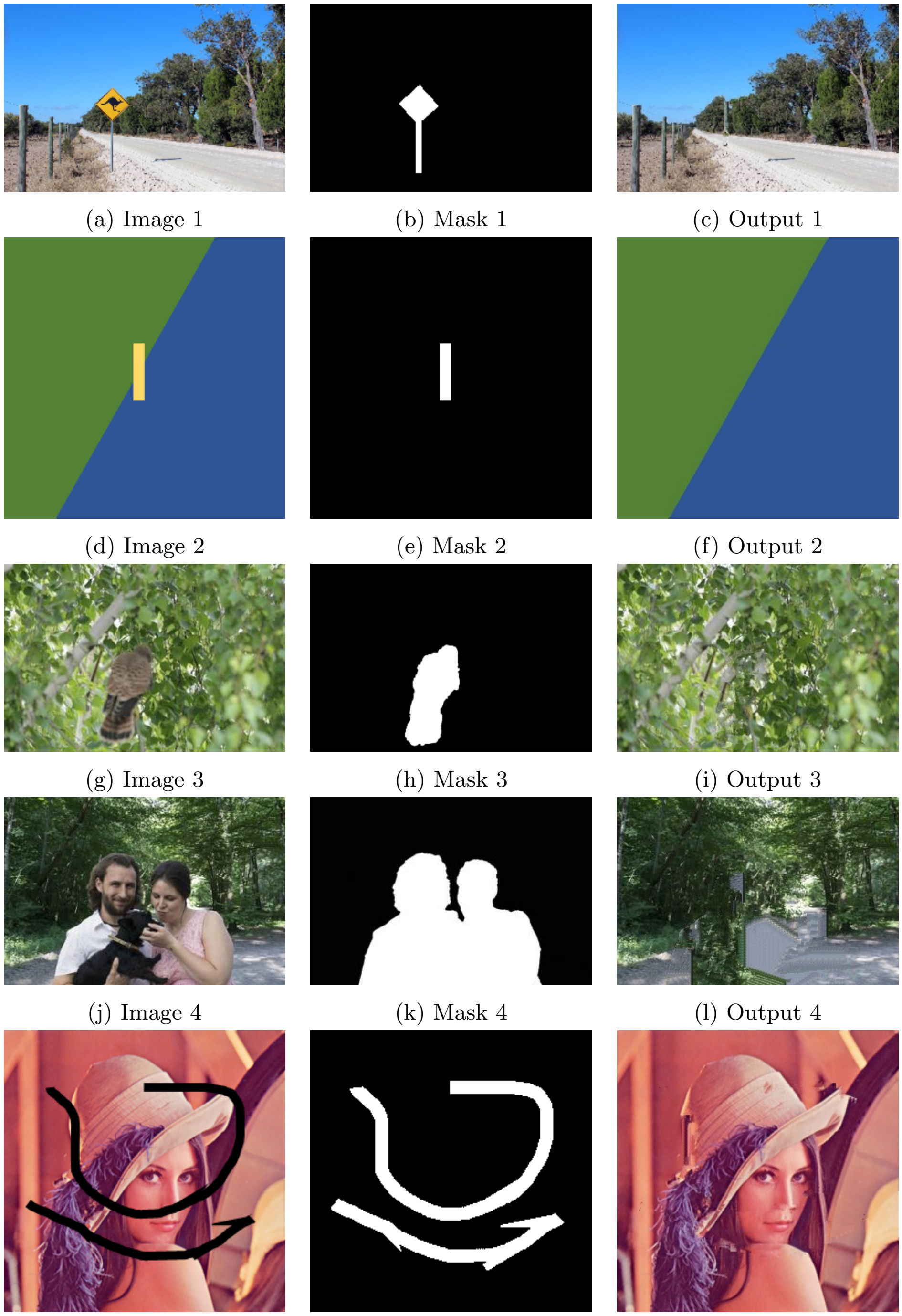


Figure 1: Notation diagram

Let I represent an image, Ω is a missing region or in other words target region to be filled in, Φ is the source region which feeds the algorithm to draw onto Ω by predicting each points p on the intersected contour line $\delta\Omega$, see Figure 1. However, choosing p for every $\delta\Omega$ step is not random but has an intelligent filling-order: p has a priority value depends on both confidence and contextual data based on image gradient. Algorithm crops a rectangle patch Ψ_p centered on p then counts the non-missing pixels in that area which resulted in confidence probability $C(p)$ after the summation is divided by $|\Psi_p|$. The data term, whereas, assigns more importance to the pixels placed in the continuation of image structures; $D(p) = |\nabla I_p^\perp \cdot n_p|/\alpha$ where $\nabla I_p^\perp$ is the isophote at p, n_p is the normal to $\delta\Omega$ and α is the maximum range of intensity levels. Priority is defined as the multiplication of confidence and data terms. Here, filling operation starts on patch Ψ_p whose p have the largest priority. Then algorithm searches same shaped rectangle areas Ψ_q on source region Φ that it has a minimum difference to Ψ_p based on squared distance between pixel values of patches and euclidean distance between their spatial locations. The one Ψ_q providing the minimum difference is used to fill the empty pixels of Ψ_p. After an iteration priorities are updated among the rest empty pixels.

Figure 2 shows the output for the test images of the algorithm explained above. The size of patches, Ψ_p and Ψ_q, are picked as 5×5.

[1] available at https://github.com/igorcmoura/inpaint-object-remover



(a) Image 1 (b) Mask 1 (c) Output 1

(d) Image 2 (e) Mask 2 (f) Output 2

(g) Image 3 (h) Mask 3 (i) Output 3

(j) Image 4 (k) Mask 4 (l) Output 4

(m) Image 5 (n) Mask 5 (o) Output 5

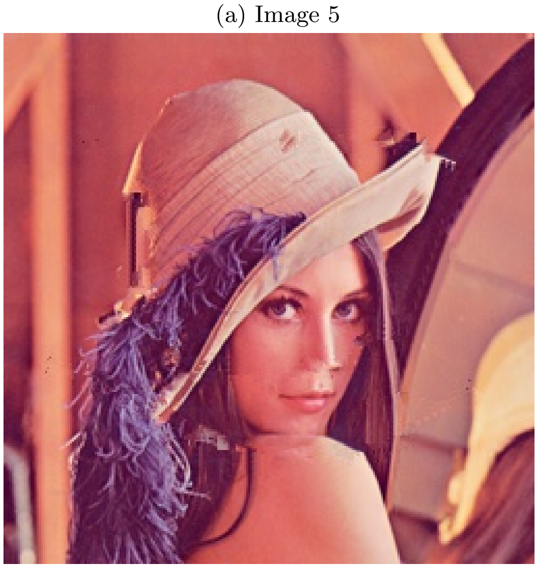
Figure 2: Applying default algorithm with 5×5 patch size

Improvement

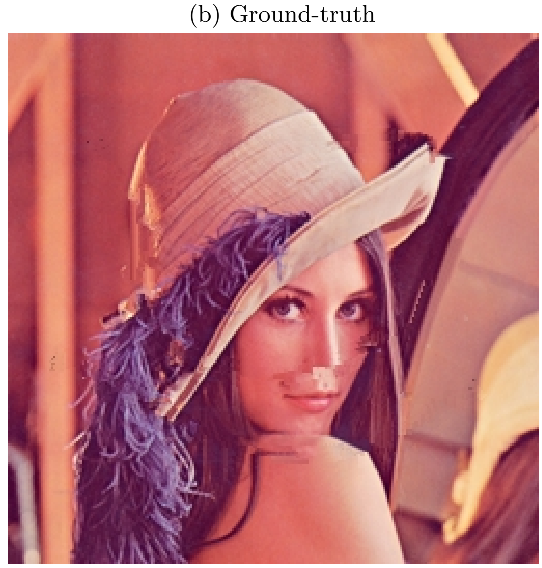
Reference algorithm output, see Figure 3 (c), struggles to find the most similar patch in source region, e.g. Lena's top left part of the hat have darker predictions however it is full-bright as seen in ground-truth (b). Main reason of this problem comes from the uncertainty of measured squared-distances of patch contents. It is not practical to expect algorithm to match a perfect patch in a single region. Moreover, finding the best exemplar in a far location is not realistic. Thus, the improved algorithm constructs a better patch by combining k number of patches together. These patches are calculated with same distance metrics yet a regularization parameter is added to penalize the spatial distance more. See improved output (d), Lena's hat color is now adjusted to the light, properly.

(a) Image 5

(b) Ground-truth



(c) Reference output (PSNR: 35.65)



(d) Improved output (PSNR: 36.53)

Figure 3: Performance of reference algorithm versus improved version. Improvement refers to updating target patch by weighted sum of k source patches, and using a regularization parameter which forces selection of source patches to be spatially closer.

Implementation of the Improvement

There are modifications in the reference project; some of slight changes are highlighted in the shared code but not showed in this report. Here is the mainly updated part and its explanation:

```
class Inpainter():
    def __init__(self, image, mask, patch_size=9, plot_progress=False,
            k=5,
            lambda_dist=100):

        # Only new attributes are displayed here
        self.k = k # number of patches combined in source region
        self.lambda_dist = lambda_dist # regularizer for euclidean distance

    # New algorithm is implemented on following method

    '''
    This method computes `squared_distance` and `euclidean_distance`
    via `self._calc_patch_difference()` method for each `source_patch`
    in the image. Computed distances are aggregated in `diff` variable
    weighted by regularization parameter `self.lambda_dist`. The most
    similar `k` source patches based on minimum `diff` scores and also
    corresponding `diff` scores are stored in `top_k_match` and `top_k_diff`
    lists, respectively. At the end of the loop, distance scores in
    `top_k_diff` list are converted to weights by inversion and normalization
    operations. Finally, the patches in `top_k_match` list are weighted sum
    to construct `source_data`.
    '''

    def _find_source_patch(self, target_pixel):
        target_patch = self._get_patch(target_pixel)
        height, width = self.working_image.shape[:2]
        patch_height, patch_width = self._patch_shape(target_patch)

        top_k_match = [] # the most similar source patches
        top_k_diff = []  # corresponding `diff` scores

        lab_image = rgb2lab(self.working_image)

        for y in range(height - patch_height + 1):
            for x in range(width - patch_width + 1):
                source_patch = [
                    [y, y + patch_height-1],
                    [x, x + patch_width-1]
                ]
                if self._patch_data(self.working_mask, source_patch) \
                   .sum() != 0:
                    continue

                squared_distance, euclidean_distance = self._calc_patch_difference(
                    lab_image,
                    target_patch,
                    source_patch
                )

                diff = squared_distance + self.lambda_dist * euclidean_distance

                diff_thr = np.max(top_k_diff) if top_k_diff else np.inf

                if diff < diff_thr or len(top_k_match) < self.k:
                    top_k_match.append(source_patch)
                    top_k_diff.append(diff)

                    if len(top_k_match) > self.k:
                        worst_idx = np.argmax(top_k_diff)
                        del top_k_match[worst_idx]
                        del top_k_diff[worst_idx]

        # diff to weight conversion
        top_k_weight = 1/np.array(top_k_diff)
        top_k_weight /= np.sum(top_k_weight)
```

```
        # initialize source data for weighted sum
        weighted_sum = np.zeros_like(
            self._patch_data(self.working_image,target_patch)
            )
        # weighted sum of `top_k_match` patch data
        for i in range(self.k):
            np.add(
                weighted_sum,
                top_k_weight[i] * self._patch_data(self.working_image,top_k_match[i]),
                out = weighted_sum,
                casting='unsafe'
            )

        return weighted_sum
```