Homework 3:
Connected Component Analysis
&
Color Correction

陳宥銓
313554013
Institute of Data Science and Engineering

Contents

1 Implementation

1.1 Task1 : Connected Component Analysis

1.1.1 Otsu's Thresholding Method

Before performing connected component analysis, we use otsu's thresholding method to binarize the image. Otsu's method is an image binarization technique that automatically determines the optimal threshold value by maximizing the variance between foreground and background of image.

1. Describe the histogram as a probability distribution by $p_i = \frac{n_i}{N}$
2. Define $a(t) = \sum_{i=0}^{t} p_i$, $b(t) = \sum_{i=t+1}^{255} p_i$,
   where t is the threshold value
3. Find t such that $\underset{t}{max}\{\frac{(m_a(t)-ma(t))^2}{a(t)b(t)}\}$,
   where $m = \sum_{i=0}^{255} ip_i$, $m_a(t) = \sum_{i=0}^{t} ip_i$

```
"""
TODO Binary transfer
"""
def to_binary(img, preprocess=0):
    ##### Otsu's thresholding method #####
    total_pixels = img.size
    hist, _ = np.histogram(img, bins=256, range=(0, 256))
    hist = hist / total_pixels # Probability distribution of gray levels

    current_max = 0
    threshold = 0
    m = np.dot(np.arange(256), hist)
    a, b, m_a = 0, 0, 0

    for t in range(256):
        a += hist[t]  # Cumulative probability of the first class (below threshold)
        b = 1 - a     # Cumulative probability of the second class (above threshold)
        m_a += t * hist[t] # Cumulative mean intensity of the first class
        if a == 0 or b == 0:
            continue
        var = ((m_a - (m * a)) ** 2) / (a * b) # Compute between-class variance
        if var > current_max: # Update the maximum variance and optimal threshold
            current_max = var
            threshold = t

    # Create a binary image using threshold
    binary_img = (img < threshold).astype(np.uint8)
    if preprocess:
        # Fillimg small region
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (7, 7))
        closed_img = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, kernel)
        return closed_img
    else:
        return binary_img
```

Figure 1: Otsu's Thresholding Method

1.1.2 Two pass Algorithm

The Two-Pass algorithm consists of two main stages: In the first pass, the image is scanned from top-left to bottom-right, and each pixel is assigned a temporary label while recording equivalence relationships between labels of neighboring pixels. In the second pass, the image is scanned again from bottom-right to top-left, and the temporary labels are updated using the equivalence information to assign unique labels to each connected component.

```python
def two_pass(binary_img, connectivity, postprocess=0):
    if connectivity == 4:
        neighbors = [(0, -1), (-1, 0)]
    else :
        neighbors = [(0, -1), (-1, 0), (-1, 1), (-1, -1)]

    rows, cols = binary_img.shape
    labels = np.zeros_like(binary_img, dtype=np.int64)
    uf = UnionFind(rows * cols)

    # First pass: Assign labels and record equivalences
    label_num = 1
    for y in range(rows):
        for x in range(cols):
            if binary_img[y, x]:
                adjacent_labels = []
                for dy, dx in neighbors: # Check neighbor's label
                    ny, nx = y + dy, x + dx
                    if 0 <= ny < rows and 0 <= nx < cols and binary_img[ny, nx] and labels[ny, nx] > 0:
                        adjacent_labels.append(labels[ny, nx]) # Collect adjacent labels

                if adjacent_labels: # If neighbors have labels
                    min_label = min(l for l in adjacent_labels)
                    labels[y, x] = min_label # Assign adjacent minimum label
                    for l in adjacent_labels:
                        if l != min_label:
                            uf.union(l, min_label) # Union other labels
                else: # Assign new label
                    labels[y, x] = label_num
                    label_num += 1

    # Second pass: Relabel
    for y in range(rows):
        for x in range(cols):
            # Set the same set to the root label
            if labels[y, x] > 0: labels[y, x] = uf.find(labels[y, x])

    # Removing small region
    if postprocess:
        labels = remove_small_region(labels, label_num, threshold=500)

    # Re-map the labels
    labels = remap_label(labels)

    return labels
```

Figure 2: Two pass Algorithm

```python
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [0] * size

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)

        if root_x != root_y:
            # Union by rank
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
```

Figure 3: Union Find

1.1.3 Seed Filling Algorithm

The Seed Filling algorithm starts from a given seed point and iteratively checks neighboring pixels to determine if they belong to the same component. If a neighbor satisfies the criteria, it is marked as part of the same component, and its neighbors are subsequently checked. This process continues until all connected pixels are visited and labeled.

```python
def seed_filling(binary_img, connectivity, postprocess=0):
    if connectivity == 4:
        neighbors = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    else :
        neighbors = [(0, 1), (1, 0), (0, -1), (-1, 0),
                     (1, 1), (1, -1), (-1, 1), (-1, -1)]

    rows, cols = binary_img.shape
    labels = np.zeros_like(binary_img, dtype=np.int64)

    label_num = 1
    for y in range(rows):
        for x in range(cols):
            if binary_img[y, x] and labels[y, x] == 0:
                stack = [(y, x)] # Initialize the seed stack
                while stack: # Filling
                    current_y, current_x = stack.pop() # Pop next adjacent labels
                    if labels[current_y, current_x] != 0: # Skip the marked pixels
                        continue
                    labels[current_y, current_x] = label_num # Assign label
                    for dy, dx in neighbors: # Check neighbors
                        ny, nx = current_y + dy, current_x + dx
                        if 0 <= ny < rows and 0 <= nx < cols and binary_img[ny, nx] and labels[ny, nx] == 0:
                            stack.append((ny, nx)) # Push adjacent labels into stack
                label_num += 1

    # Removing small region
    if postprocess:
        labels = remove_small_region(labels, label_num, threshold=500)
        # Re-map the labels
        labels = remap_label(labels)

    return labels
```

Figure 4: Seed Filling Algorithm

1.1.4 Run-Length Encoding (RLE)-Based Methods

Run-Length Encoding (RLE)-Based Methods in the context of connected component analysis leverage the concept of compressing rows of a binary image into "runs (start_position, end_position)" of consecutive pixels with the same value (Fig.5).
For example,

Input row :	0 0 1 1 1 0 0 1 1 0 0
Encoded runs :	[(2, 4), (7, 8)]

After compressing each row of the binary image into runs, neighboring runs from consecutive rows are linked if they overlap horizontally or touch diagonally. During the first pass, each run is assigned a label, then apply Union-Find to unify labels that belong to the same connected component.

```python
def rle_encode(binary_img):
    runs = []
    rows, cols = binary_img.shape
    for row in range(rows):
        start = None
        for col in range(cols):
            if binary_img[row, col] == 1:
                if start is None:
                    start = col # Start of a run
            elif start is not None:
                runs.append((row, start, col - 1)) # End of a run
                start = None
        if start is not None:  # Handle runs that end at the row's edge
            runs.append((row, start, cols - 1))
    return runs
```

Figure 5: Run-Length Encoding

```
"""
Run-Length Encoding (RLE)-Based Methods
"""
def other_cca_algorithm(binary_img, connectivity, postprocess=0):
    runs = rle_encode(binary_img) # Encode the image into runs
    uf = UnionFind(len(runs))
    run_labels = {}  # Store the labels for each run
    current_label = 1

    for i, run in enumerate(runs):
        row, start, end = run
        neighbors = [] # Neighbors from the previous row

        # Check for neighbors in the previous row
        for j in range(i): # Only consider the previous row
            prev_row, prev_start, prev_end = runs[j]
            if prev_row == row - 1:
                # 8-connectivity: check horizontal and diagonal
                if connectivity == 8 and max(start, prev_start-1) <= min(end, prev_end+1):
                    neighbors.append(j)
                # 4-connectivity: check horizontal
                if connectivity == 4 and max(start, prev_start) <= min(end, prev_end):
                    neighbors.append(j)
```

```
        # Assign label to the current run
        if not neighbors:  # No neighbors, assign a new label
            run_labels[i] = current_label
            current_label += 1
        else: # Use the smallest label among neighbors
            first_label = run_labels[neighbors[0]]
            run_labels[i] = first_label
            for neighbor in neighbors:
                uf.union(first_label, run_labels[neighbor])

    # Compress labels to their roots
    for i in run_labels:
        run_labels[i] = uf.find(run_labels[i])

    # Write labels back to the image
    labeled_image = np.zeros_like(binary_img, dtype=np.int32)
    for i, run in enumerate(runs):
        row, start, end = run
        labeled_image[row, start:end + 1] = run_labels[i]

    if postprocess:
        labels = remove_small_region(labeled_image, current_label, threshold=500)
        # Re-map the labels
        labels = remap_label(labels)
    return labels
```

Figure 6: Run-Length Encoding Methods for CCA

1.1.5 Color mapping

To perform color mapping, we first remapping the labels to continuous numbers (Fig.7), then creating a set to store unique colors, ensuring that there is one color for each label in the input image. It generates random RGB colors until the set has as many unique colors as there are labels (from 0 to the maximum label value in the image) and the background (label 0) color in the set is set to black (Fig.8).

```python
def remap_label(label_img):
    rows, cols = label_img.shape
    # Re-map the labels
    unique_labels = np.unique(label_img)
    label_map = {label: idx for idx, label in enumerate(unique_labels)}
    for y in range(rows):
        for x in range(cols):
            label_img[y, x] = label_map[label_img[y, x]]
    return label_img
```

Figure 7: Remapping the Labels

```python
def color_mapping(label_img):
    color_map = set()
    while len(color_map) < np.max(label_img) + 1:
        color = tuple(np.random.randint(0, 256, size=3))
        color_map.add(color)
    color_map = list(color_map)
    color_map[0] = [0,0,0]

    rows, cols = label_img.shape
    color_image = np.zeros((rows, cols, 3), dtype=np.uint8)
    for y in range(rows):
        for x in range(cols):
            color_image[y, x] = color_map[label_img[y, x]]
    return color_image
```

Figure 8: Color Mapping

1.2 Task2 : Color Correction

1.2.1 White Patch Algorithm

The white patch algorithm assumes that the brightest pixels in the image are white and adjusts the color balance of the image based on that assumption. The algorithm is as follows

1. Extract maximum value R_{max}, G_{max}, B_{max} in RGB channel
2. Compute the normalize factor
   $R_{normal} = \frac{255}{R_{max}}$, $G_{normal} = \frac{255}{G_{max}}$, $B_{normal} = \frac{255}{B_{max}}$
3. Adjust pixels value
   $R' = R_{normal} * R_{max}$, $G' = G_{normal} * G_{max}$, $B' = B_{normal} * B_{max}$

```python
def white_patch_algorithm(img):
    B, G, R = cv2.split(img.astype('float')) # Split the channel
    R_max, G_max, B_max = np.max(R), np.max(G), np.max(B) # Maximum of each channel
    n_R, n_G, n_B = 255 / R_max, 255 / G_max, 255 / B_max # Normalize
    R = np.clip(R * n_R, 0, 255) # Balance
    G = np.clip(G * n_G, 0, 255)
    B = np.clip(B * n_B, 0, 255)
    balanced_image = cv2.merge([B, G, R]).astype('uint8')
    return balanced_image
```

Figure 9: White Patch Algorithm

1.2.2 Gray-world Algorithm

The gray world algorithm assumes that the means of the RGB channels are equal and adjusts the mean of each channel to achieve white balance. The algorithm is as follows

1. Compute the mean of each channel μ_R, μ_G, μ_B
2. Compute the gray value $g = \frac{\mu_R + \mu_G + \mu_B}{3}$
3. Compute the normalize factor
   $R_{normal} = \frac{g}{\mu_R}$, $G_{normal} = \frac{g}{\mu_G}$, $B_{normal} = \frac{g}{\mu_B}$
4. Adjust pixels value
   $R' = R_{normal} * R$, $G' = G_{normal} * G$, $B' = B_{normal} * B$

```python
def gray_world_algorithm(img):
    B, G, R = cv2.split(img.astype('float'))
    R_mean, G_mean, B_mean= np.mean(R), np.mean(G), np.mean(B) # Mean of each channel
    avg = (R_mean + G_mean + B_mean) / 3
    n_R, n_G, n_B = avg / R_mean, avg / G_mean, avg / B_mean # Normalize
    R = np.clip(R * n_R, 0, 255) # Balance
    G = np.clip(G * n_G, 0, 255)
    B = np.clip(B * n_B, 0, 255)
    balanced_image = cv2.merge([B, G, R]).astype('uint8')
    return balanced_image
```

Figure 10: Gray-world Algorithm

1.2.3 Shades of Gray Algorithm

The shades of gray algorithm combines the gray world and the white patch algorithm to adapt to different scenarios by adjusting the parameter p. The algorithm is as follows

1. Compute the mean μ_C of each channel where
   $\mu_C = (\frac{1}{N}\sum_{i=1}^{N} C_i^p)^{\frac{1}{p}}, C = R, G, B$
2. Compute the gray value $g = \frac{\mu_R + \mu_G + \mu_B}{3}$
3. Compute the normalize factor
   $R_{normal} = \frac{g}{\mu_R}$, $G_{normal} = \frac{g}{\mu_G}$, $B_{normal} = \frac{g}{\mu_B}$
4. Adjust pixels value
   $R' = R_{normal} * R$, $G' = G_{normal} * G$, $B' = B_{normal} * B$

```python
def other_white_balance_algorithm(img, p=6):
    """
    Reference:
    Finlayson, Graham D., et al.
    "Shades of gray and colour constancy."
    """
    B, G, R = cv2.split(img.astype('float')) # Split the channel
    R_norm = np.power(np.mean(np.power(R, p)), 1/p)
    G_norm = np.power(np.mean(np.power(G, p)), 1/p)
    B_norm = np.power(np.mean(np.power(B, p)), 1/p)
    avg = (R_norm + R_norm + R_norm) / 3
    n_R = avg / R_norm
    n_G = avg / G_norm
    n_B = avg / B_norm

    R = np.clip(R * n_R, 0, 255)
    G = np.clip(G * n_G, 0, 255)
    B = np.clip(B * n_B, 0, 255)
    balanced_image = cv2.merge([B, G, R]).astype('uint8')
    return balanced_image
```

Figure 11: Shades of Gray Algorithm

2 Results and Analysis

2.1 Task1 : Connected Component Analysis

Fig.12 and Fig.13 show the results of CCA with 4-connectivity and 8-connectivity using different methods. We found that although 4-connectivity only considers the pixel relationship between up, down, left and right, and the calculation load is relatively small, it may not be fully connected for objects with more complex shapes (such as diagonal lines or curves). On the other hand, 8-connectivity is suitable for processing complex shapes and more accurate connectivity analysis, ensuring better connectivity, especially for diagonally connected areas. But it may lead to misconnections and has higher computational complexity. Tab. 1 shows the execution time of each method. We observed that the RLE-based method performs the least time compared to other methods, demonstrating its efficiency. This advantage can be attributed to the reduced amount of data processed, as it compresses consecutive pixels into runs, significantly lowering the computational complexity. In contrast, other methods require a pixel-by-pixel search, which is computationally more expensive, especially for large images with many connected components.

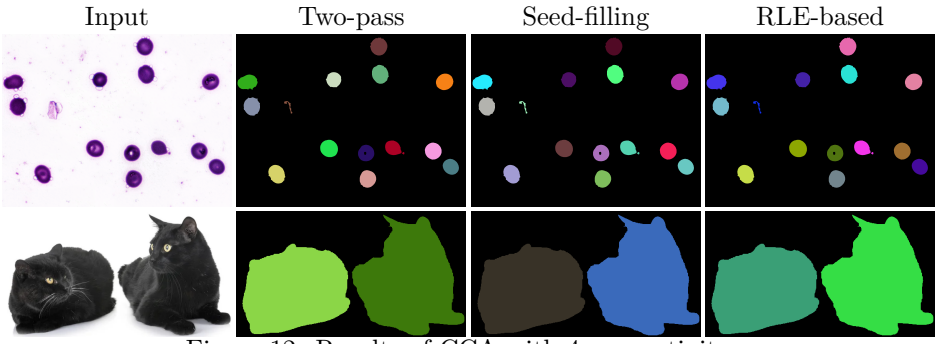

Figure 12: Results of CCA with 4-connectivity

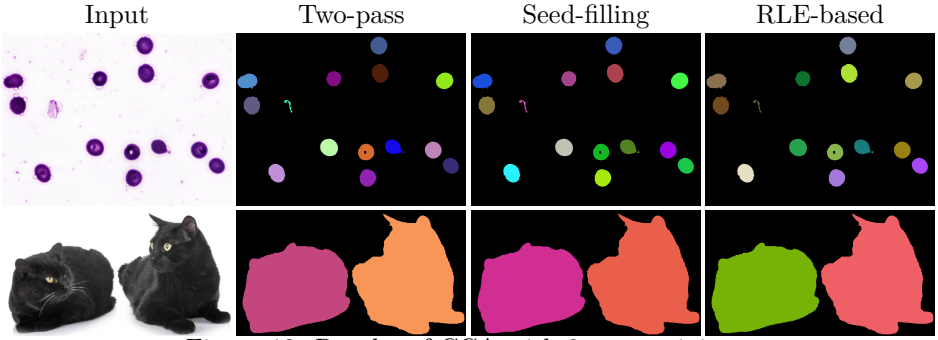

Figure 13: Results of CCA with 8-connectivity

Image1	Two-pass	Seed-filling	RLE-based
4-conn.	1.19	0.65	0.55
8-conn.	1.12	0.84	0.55
Image2	Two-pass	Seed-filling	RLE-based
4-conn.	0.20	0.16	0.04
8-conn.	0.26	0.28	0.04

Table 1: Execution Time of Different Methods

2.2 Task2 : Color Correction

Fig.14 shows the results using different white balance methods. We found that the Shades of Gray Algorithm performs better than other algorithms for **Image 1**. However, the white patch algorithm with the least satisfactory performance in **Image 1** performs the best in **Image 2**. This may be because there is no white in the **Image 1** , but a bright yellow that is similar to white. The algorithm cannot accurately balance the colors in different areas, resulting in erroneous adjustments.

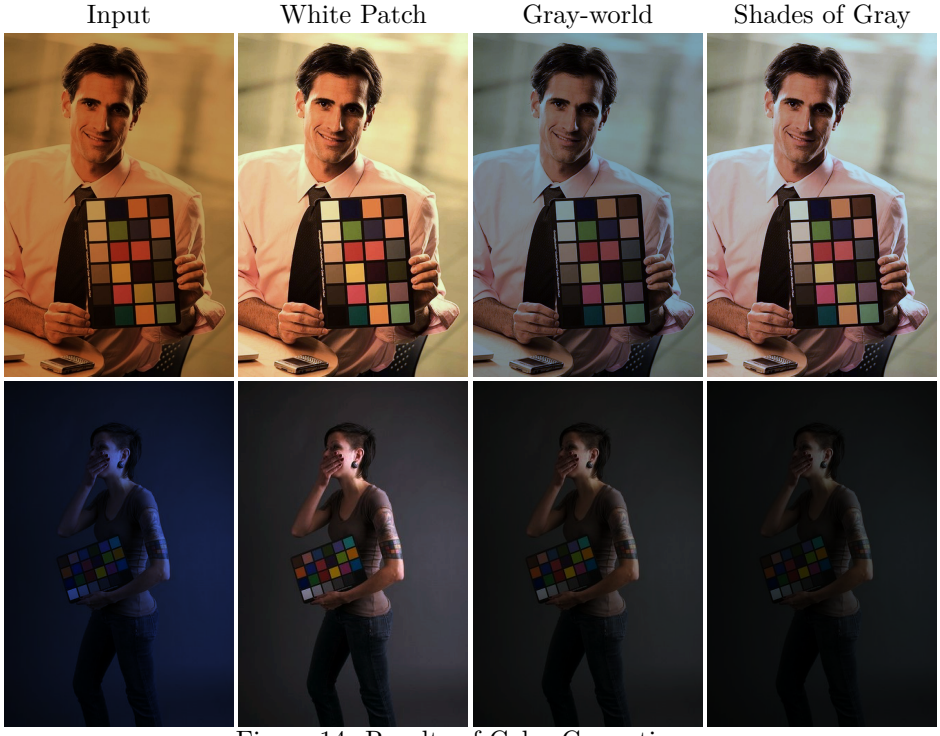

Figure 14: Results of Color Correction

3 Answer the questions

1. **Please describe a problem you encountered and how you solved it.**
   When performing connected component analysis, we found that there might be some noise and holes in the binary image, leading to unsatisfactory results. To address this, we apply following techniques

   - Fill small holes through close operation
   - Define a threshold. If the unconnected area is smaller than the threshold, set the label to 0.

```python
def remove_small_region(label_img, label_num, threshold=500):
    # Removing small region
    for i in range(1, label_num):
        if len(label_img[label_img==i]) < threshold: # Remove small region
            label_img[label_img == i] = 0
    return label_img
```

Figure 15: Remove Small Region

(a) Before (b) After

2. **What are the advantages and limitations of two-pass and seed-filling algorithms for object segmentation in images, and in which scenarios are they most appropriate?**
   These algorithms can typically process images efficiently, especially in images that are small in size or have simple structures. For target objects with clear connectivity, they can accurately segment the target areas. However, they still face challenges, such as requiring a significant amount of memory when processing high-resolution images. For example, two-pass algorithm needs to store the intermediate data and equivalence tables.

3. **What are the advantages and limitations of the white patch and gray-world algorithms for image white balance, and in which scenarios are they most appropriate?** The white patch algorithm is easy to implement, has low computational cost, and is suitable for real-time applications. It provides accurate color correction when there are true white areas in the image. However, due to the reliance on the white point, an outlier pixel or noise may affect the effect of the entire white balance

operation. If the maximum value of each channel in the original image cannot represent the white light information, it may lead to unsatisfactory color results. Gray-World Algorithm is calculated based on the global average color. It is less sensitive to noise and does not require white areas as a reference. It is suitable for images with a color distribution close to neutral. However, when the color in the image is relatively uniform or the average color deviates significantly from gray, it will cause the algorithm to produce inaccurate white balance adjustments.