Homework 1: Connected Component Analysis & Color Correction

Part I. Implementation (5%):

Task 1: Connected Component Analysis

To_binary: First transfer the RGB image to Grayscale image, and then transfer the Grayscale image to Binary image. If the pixel value is larger than 127, set it to 0. Otherwise, set the value to 255.

```python
"""
TODO Binary transfer
"""
def to_binary(img):
    if img is None:
        print('Can not load the image.')
    else:
        grayscale_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        threshold_value = 127
        _, binary_image = cv2.threshold(grayscale_image, threshold_value, 255, cv2.THRESH_BINARY_INV)

        return binary_image
```

Union & Find: Use to implement Two-pass algorithm and Other algorithm

```python
def find(parent, i):
    if parent[i] == i:
        return i
    parent[i] = find(parent, parent[i]) # path compression
    return parent[i]

def union(parent, i, j):
    root_i = find(parent, i)
    root_j = find(parent, j)
    if root_i != root_j:
        parent[root_j] = root_i
```

Color Mapping: Create rainbow color to color-map the image. Each segment will be set a color.

```python
"""
TODO Color mapping
"""
def color_mapping(labeled_image):
    """
    color mapping

    input:
        labeled_image (np.array): the image which is labeled

    output:
        np.array: a 3d NumPy array representing the color-mapped image
    """
    unique_labels = np.unique(labeled_image)
    unique_labels = unique_labels[unique_labels != 0]

    if len(unique_labels) == 0:
        return np.zeros((*labeled_image.shape, 3), dtype=np.uint8)

    color_image = np.zeros((*labeled_image.shape, 3), dtype=np.uint8)
    num_labels = len(unique_labels)

    hue_values = np.linspace(0, 179, num_labels)

    hsv_palette = np.zeros((num_labels, 1, 3), dtype=np.uint8)
    hsv_palette[:, 0, 0] = hue_values
    hsv_palette[:, 0, 1] = 255
    hsv_palette[:, 0, 2] = 255

    bgr_palette = cv2.cvtColor(hsv_palette, cv2.COLOR_HSV2BGR)

    label_to_color = {label: bgr_palette[i][0].tolist()
                      for i, label in enumerate(unique_labels)}

    for label, color in label_to_color.items():
        color_image[labeled_image == label] = color

    return color_image
```

Main: Add another makedirs and imwrite for bonus algorithm

```python
"""
Main function
"""
def main():

    os.makedirs("result/connected_component/two_pass", exist_ok=True)
    os.makedirs("result/connected_component/seed_filling", exist_ok=True)
    os.makedirs("result/connected_component/block_based", exist_ok=True)
    connectivity_type = [4, 8]

    for i in range(2):
        img = cv2.imread("data/connected_component/input{}.png".format(i + 1))

        for connectivity in connectivity_type:

            # TODO Part1: Transfer to binary image
            binary_img = to_binary(img)

            # TODO Part2: CCA algorithm
            two_pass_label = two_pass(binary_img, connectivity)
            seed_filling_label = seed_filling(binary_img, connectivity)
            block_based_label = other_cca_algorithm(binary_img, connectivity, (64, 64))

            # TODO Part3: Color mapping
            two_pass_color = color_mapping(two_pass_label)
            seed_filling_color = color_mapping(seed_filling_label)
            block_based_color = color_mapping(block_based_label)

            cv2.imwrite("result/connected_component/two_pass/input{}_c{}.png".format(i + 1, connectivity), two_pass_color)
            cv2.imwrite("result/connected_component/seed_filling/input{}_c{}.png".format(i + 1, connectivity), seed_filling_color)
            cv2.imwrite("result/connected_component/block_based/input{}_c{}.png".format(i + 1, connectivity), block_based_color)

if __name__ == "__main__":
    main()
```

- Two-pass Algorithm

```python
def two_pass(binary_image, connectivity):
    """
    2-pass CCA for 4 and 8 connectivity

    input:
        binary_image (np.array): A 2D NumPy array
        connectivivity (int): 4 or 8

    output:
        np.array: The labeled image
    """
    rows, cols = binary_image.shape
    labeled_image = np.zeros((rows, cols), dtype=np.int32)
    next_label = 1
    parent = [0]
```

First Pass: From top-left of the image, give all the pixels the same label if they are neighbors. After go through the entire images, you can find all the labeled segments.

The program test the top-left, top-right, top and left pixel to see if it is neighbor of the target pixel if the connectivity is 8. If the connectivity is 4, we just check top and left pixel.

```python
    # First Pass
    for y in range(rows):
        for x in range(cols):
            if binary_image[y, x] == 255:
                neighbors = []

                if connectivity == 8:
                    # top-left
                    if y > 0 and x > 0 and labeled_image[y-1, x-1] > 0:
                        neighbors.append(labeled_image[y-1, x-1])
                    # top-right
                    if y > 0 and x < cols - 1 and labeled_image[y-1, x+1] > 0:
                        neighbors.append(labeled_image[y-1, x+1])

                # top
                if y > 0 and labeled_image[y-1, x] > 0:
                    neighbors.append(labeled_image[y-1, x])
                # left
                if x > 0 and labeled_image[y, x-1] > 0:
                    neighbors.append(labeled_image[y, x-1])

                if not neighbors:
                    labeled_image[y, x] = next_label
                    parent.append(next_label)
                    next_label += 1
                else:
                    neighbor_roots = [find(parent, label) for label in neighbors]

                    min_label = min(neighbor_roots)
                    labeled_image[y, x] = min_label

                    for label in neighbors:
                        union(parent, min_label, label)
```

Second Pass: Union and Find the labeled image. To union the part which should be the same label.

```python
    # Second pass
    for y in range(rows):
        for x in range(cols):
            if labeled_image[y, x] > 0:
                original_label = labeled_image[y, x]
                final_label = find(parent, original_label)
                labeled_image[y, x] = final_label

    return labeled_image
```

- Seed-filling Algorithm

First Part: Set some parameters.

```python
"""
TODO Seed filling algorithm
"""
def seed_filling(binary_image, connectivity):
    """
    seed_filling for 4 and 8 connectivity

    input:
        binary_image (np.array): A 2D NumPy array
        connectivity (int): 4 or 8

    output:
        np.array: The labeled image
    """
    rows, cols = binary_image.shape
    labeled_image = np.zeros((rows, cols), dtype=np.int32)
    next_label = 1

    if connectivity == 4:
        neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    elif connectivity == 8:
        neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1),
                     (-1, -1), (-1, 1), (1, -1), (1, 1)]
```

Second Part: Find the first pixel value which is 255. Add in it the queue, and then use BFS to find the neighbors.

```python
    for y in range(rows):
        for x in range(cols):
            if binary_image[y, x] == 255 and labeled_image[y, x] == 0:
                queue = [(y, x)]
                labeled_image[y, x] = next_label

                while queue:
                    py, px = queue.pop(0)

                    for dy, dx in neighbors:
                        ny, nx = py + dy, px + dx

                        if 0 <= ny < rows and 0 <= nx < cols and binary_image[ny, nx] == 255 and labeled_image[ny, nx] == 0:
                            labeled_image[ny, nx] = next_label
                            queue.append((ny, nx))

                next_label += 1

    return labeled_image
```

- (Bonus) Other Algotithms: Block-Based Algorithm. Use Divide & Conquer to make it fast

Divide: Divide the image to blocks with the tile_size

```python
"""
Bonus
"""
def other_cca_algorithm(binary_image, connectivity, tile_size=(64, 64)):
    """
    Block-based CCA. Use Divide & Conquer and 2-pass to solve the CCA problem

    Input:
        binary_image (np.array): A 2D NumPy array
        connectivity (int): 4 or 8
        tile_size (int, int): The block size

    Output:
        np.array: A labeled image
    """
    rows, cols = binary_image.shape
    t_rows, t_cols = tile_size

    # 1. Divide
    num_tiles_y = (rows + t_rows - 1) // t_rows
    num_tiles_x = (cols + t_cols - 1) // t_cols

    labeled_tiles = {}
    label_offset = 0
```

Conquer: Label each pixels from each blocks

```python
    # 2. Conquer
    for i in range(num_tiles_y):
        for j in range(num_tiles_x):
            y_start, x_start = i * t_rows, j * t_cols
            tile = binary_image[y_start : y_start + t_rows, x_start : x_start + t_cols]

            labeled_tile= two_pass(tile, connectivity)

            non_zero_mask = labeled_tile > 0
            if np.any(non_zero_mask):
                labeled_tile[non_zero_mask] += label_offset
                label_offset = np.max(labeled_tile)

            labeled_tiles[(i, j)] = labeled_tile
```

Merge: Go through the vertical of the egde of the block and see if the label are the same as the left edge of the other block. If yes, merge them. Do the same thing with the horizon.

```python
    # 3. Merge
    parent = list(range(label_offset + 1)) # Initialize parent array for Union-Find

    for i in range(num_tiles_y):
        for j in range(num_tiles_x):
            tile = labeled_tiles[(i, j)]

            # Merge with the tile to the right
            if j < num_tiles_x - 1:
                right_tile = labeled_tiles[(i, j + 1)]
                for y_idx in range(min(tile.shape[0], right_tile.shape[0])):
                    if tile[y_idx, -1] > 0 and right_tile[y_idx, 0] > 0:
                        union(parent, tile[y_idx, -1], right_tile[y_idx, 0])

            # Merge with the tile below
            if i < num_tiles_y - 1:
                bottom_tile = labeled_tiles[(i + 1, j)]
                for x_idx in range(min(tile.shape[1], bottom_tile.shape[1])):
                    if tile[-1, x_idx] > 0 and bottom_tile[0, x_idx] > 0:
                        union(parent, tile[-1, x_idx], bottom_tile[0, x_idx])
```

Final Relabeling: Use Union & Find to union the part which should be the same label.

```python
    # 4. Final Relabeling
    final_labeled_image = np.zeros((rows, cols), dtype=np.int32)
    for i in range(num_tiles_y):
        for j in range(num_tiles_x):
            y_start, x_start = i * t_rows, j * t_cols
            tile = labeled_tiles[(i, j)]

            h, w = tile.shape

            # Find the root for each label in the tile
            unique_labels = np.unique(tile[tile > 0])
            for label in unique_labels:
                root = find(parent, label)
                tile[tile == label] = root

            final_labeled_image[y_start : y_start + h, x_start : x_start + w] = tile

    return final_labeled_image
```

Task 2: Color Correction

- White Patch Algorithm: Find the 99% r, g, b pixel to correct the color. Noted: I used to us-e the maximum value, but the result of 99% is better.

```python
"""
TODO White patch algorithm
"""
def white_patch_algorithm(img):
    """
    white_patch_algorithm for color correction

    Input:
        img (np.array): A 2D NumPy array

    output:
        np.array: a color corrected image
    """
    b, g, r = cv2.split(img)

    percentile = 99
    b_max = np.percentile(b, percentile)
    g_max = np.percentile(g, percentile)
    r_max = np.percentile(r, percentile)
    # g_max = g.max()
    # r_max = r.max()
    # b_max = b.max()

    b_gain = 255.0 / b_max if b_max > 0 else 1 # avoid to divide 0
    g_gain = 255.0 / g_max if g_max > 0 else 1
    r_gain = 255.0 / r_max if r_max > 0 else 1

    b_corrected = np.clip(b.astype(np.float64) * b_gain, 0, 255)
    g_corrected = np.clip(g.astype(np.float64) * g_gain, 0, 255)
    r_corrected = np.clip(r.astype(np.float64) * r_gain, 0, 255)

    corrected_img = cv2.merge((b_corrected, g_corrected, r_corrected)).astype(np.uint8)

    return corrected_img
```

- Gray-world Algorithm: First get the average value of r, g, b, and use then to calculate the c-orrected color

```python
TODO Gray-world algorithm
"""
def gray_world_algorithm(img):
    """
    gray_world_algorithm for color correction

    Input:
        img (np.array): A 2D NumPy array

    output:
        np.array: a color corrected image
    """
    b, g, r = cv2.split(img)

    b_avg = np.mean(b)
    g_avg = np.mean(g)
    r_avg = np.mean(r)

    gray_avg = (b_avg + g_avg + r_avg) / 3.0

    b_gain = gray_avg / b_avg if b_avg > 0 else 1 # avoid to divide 0
    g_gain = gray_avg / g_avg if g_avg > 0 else 1
    r_gain = gray_avg / r_avg if r_avg > 0 else 1

    b_corrected = np.clip(b.astype(np.float64) * b_gain, 0, 255)
    g_corrected = np.clip(g.astype(np.float64) * g_gain, 0, 255)
    r_corrected = np.clip(r.astype(np.float64) * r_gain, 0, 255)

    corrected_img = cv2.merge((b_corrected, g_corrected, r_corrected)).astype(np.uint8)

    return corrected_img
```

- (Bonus) Other Algotithms

Tool (find_white_patch.py): Use this program to find the area in the image that is used as the white color reference.

```python
import cv2

def find_coordinates(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWN:
        print(f"Clicked at (x, y): ({x}, {y})")

# --- Main Part ---
image_path = 'data/color_correction/input2.bmp'
img = cv2.imread(image_path)

if img is None:
    print(f"Error: Could not load image at {image_path}")
else:
    cv2.imshow('Image - Click to find coordinates', img)
    cv2.setMouseCallback('Image - Click to find coordinates', find_coordinates)

    print("Click on the corners of the white patch. Press any key to exit.")
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

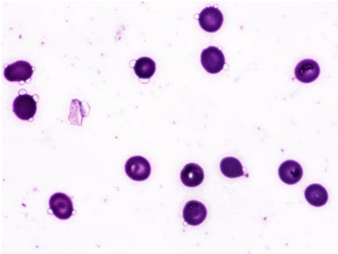
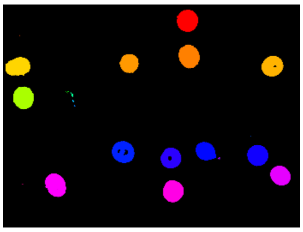
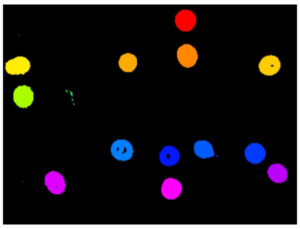
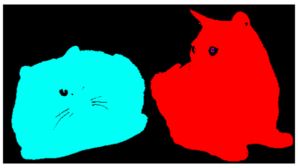
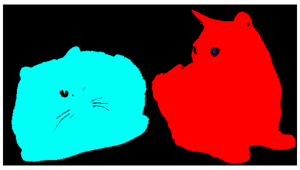
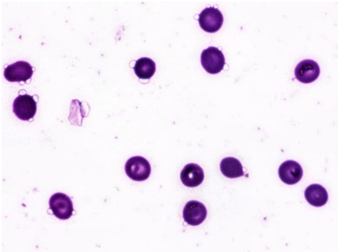
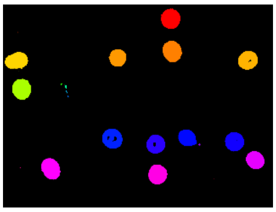
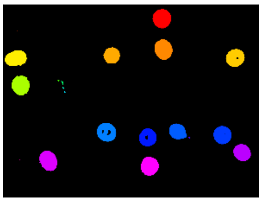
Main Part: Use the found white patch as a reference to correct the color

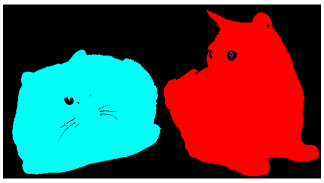
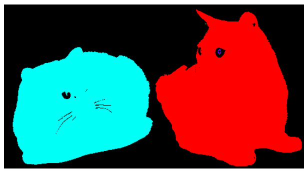
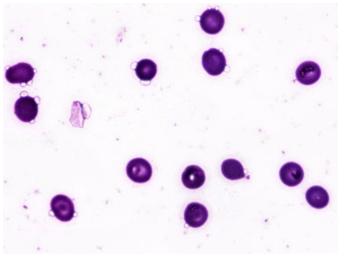
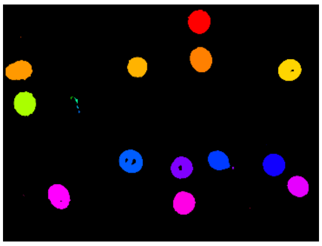
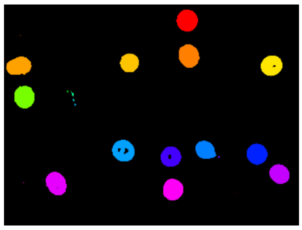
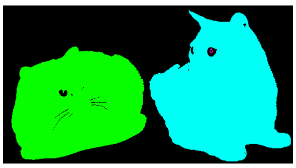
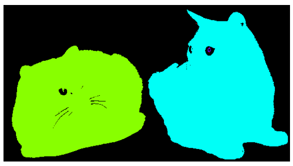
```python
"""
Bonus
"""
def other_white_balance_algorithm(img, white_patch_roi):
    """
    White Point Correction
    Corrects the color cast of an image using a reference white patch.

    Input:
        img (np.array): A 2D NumPy array image need to be corrected.
        white_patch_roi (tuple): A tuple of slices defining the white patch
                                 (y_start:y_end, x_start:x_end).

    Output:
        np.array: The color-corrected BGR image.
    """
    y_start, y_end = white_patch_roi[0].start, white_patch_roi[0].stop
    x_start, x_end = white_patch_roi[1].start, white_patch_roi[1].stop
    patch = img[y_start:y_end, x_start:x_end]

    b_avg, g_avg, r_avg = patch.mean(axis=(0,1))

    gray_avg = (b_avg + g_avg + r_avg) / 3

    b_gain = gray_avg / b_avg if b_avg > 0 else 1 # avoid to divide 0
    g_gain = gray_avg / g_avg if g_avg > 0 else 1
    r_gain = gray_avg / r_avg if r_avg > 0 else 1

    b, g, r = cv2.split(img)
    b_corrected = np.clip(b.astype(np.float64) * b_gain, 0, 255)
    g_corrected = np.clip(g.astype(np.float64) * g_gain, 0, 255)
    r_corrected = np.clip(r.astype(np.float64) * r_gain, 0, 255)

    corrected_img = cv2.merge((b_corrected, g_corrected, r_corrected)).astype(np.uint8)

    return corrected_img
```

Part II. Results & Analysis (20%):

Please provide your observations and analysis for each of the following bullets.

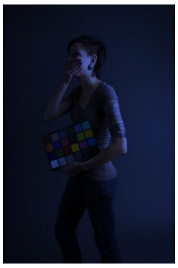
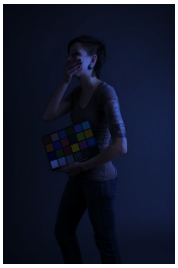
Task 1: Connected Component Analysis

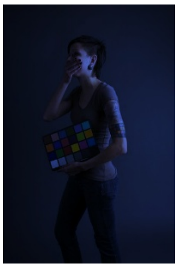
- Two-pass Algorithm
  - original image: 
  - 4-connectivity:  8-connectivity: 
  - original image: 
  - 4-connectivity:  8-connectivity: 
- Seed-filling Algorithm
  - original image: 
  - 4-connectivity:  8-connectivity: 

  - original image: 
  - 4-connectivity:  8-connectivity: 
- (Bonus) Other Algotithms
  - original image: 
  - 4-connectivity:  8-connectivity: 
  - original image: 
  - 4-connectivity:  8-connectivity: 
- Compare and discuss the above result.
  - The results does not have major difference of the three algorithm, but the bonus algorithm which use divide & conquer should be the fastest. It is because with divide & conquer, we can parrallel the program with multiple CPUs.

○

Task 2: Color Correction

- White Patch Algorithm
    - original image:  corrected image: 
    - original image:  corrected image: 
- Gray-world Algorithm
    - original image:  corrected image: 
    - original image:  corrected image: 
- (Bonus) Other Algotithms
    - original image:  corrected image: 

- original image:  corrected image: 

- Compare and discuss the above result.
  - The result of white patch is the best. I used to get the maximum rgb value in the w hith_patch_algorithm, but I got better result with 99% value.
  - The bonus algorithm pick the area of white color reference, so the result of it shoul -d be the best. I think the reason why it is not the best is I do not pick the area accu -rately.

Part III. Answer the questions (5%):

1. Please describe a problem you encountered and how you solved it.
   Ans: In the bonus algo of CCA, at first each component is colored by many colors. It is be cause I put the for-loop in the wrong position. It took much time to find it.
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?
   Ans:
   The two-pass algo scans an image twice. It is comprehensive and reliable but can be slowe r and use more memory.
   The seed-filling algo is conceptually simple but can be inefficient for images with many s mall objects. It is best for interactive tasks or when only a few specific components need t o be labeled.
3. What are the advantages and limitations of the **white patch** and **gray-world algorithms** for i mage white balance, and in which scenarios are they most appropriate?
   Ans:
   The whith patch algo assumes the brightest pixel in the image is white. It is effective for sc enes which the brightest pixel is actually white but fails for the brightest object is actually c olored, making it best for studio shots.
   The gray-world algo assumes the average of the entire scene is gray. It works well for imag es with a wide variety of colors, like landscape, but fails on images dominated by a single c olr. It is appropriate for scenes with a balanced color distribution.