Homework 3: Connected Component Analysis & Color Correction

> **Name:** 朱驛庭
> **Student ID:** 111550093

Part I. Implementation

Binary Transfer

```python
"""
TODO Binary transfer
"""
def to_binary(img, threshold=128):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return np.where(img > threshold, 0, 255).astype(np.uint8)
```

I used OpenCV's `cvtColor` function to first convert the RGB image to a grayscale image. Then, I applied `np.where` to turn pixels with brightness greater than a threshold into black, while the remaining pixels were converted to white. The reason for this approach is that both input images have white backgrounds with only a small portion of black subjects. By inverting the colors first, it facilitates the subsequent segmentation process.

Two Pass Algorithm

```python
def find_root(labels, i):
    while labels[i] != i:
        i = labels[i]
    return i

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

To facilitate the operation of the two-pass algorithm, I implemented the union-find data structure to manage label equivalences efficiently. This data structure is particularly useful for handling the merging and retrieval of connected components. It consists of two key functions:

1. **find_root**: Given a label, this function traces back through the parent nodes to find the root of the label's equivalence class. To optimize performance, path compression is applied, which flattens the structure, ensuring future queries are faster.
2. **union**: This function merges the equivalence classes of two labels by connecting their root nodes. Union by rank or size is used to keep the tree structure balanced, minimizing the depth and improving efficiency.

By leveraging these functions, the algorithm can efficiently resolve label equivalences during the second pass, ensuring all connected components are correctly identified and labeled. This approach significantly reduces computational overhead compared to simpler methods, especially when dealing with large images or complex connectivity patterns.

```python
def fill_holes(img):
    contours, _ = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    for contour in contours:
        cv2.drawContours(img, [contour], -1, (255), thickness=cv2.FILLED)
    return img
```

Before starting the two-pass algorithm, I used OpenCV's `findContours` and `drawContours` functions to fill in small holes in the image. This preprocessing step helps ensure that the connected components are better defined, reducing the complexity of segmentation later on.

Specifically:

1. `findContours`: This function detects the contours of regions in the binary image. By identifying the boundaries of small holes, we can isolate them for further processing.
2. `drawContours`: After detecting the contours, this function is used to fill in these holes by drawing over them with the desired color (e.g., white).

This preprocessing step is particularly beneficial in scenarios where small gaps or holes in objects might otherwise result in fragmented labels during the segmentation process. By addressing these inconsistencies upfront, the two-pass algorithm can produce cleaner and more accurate results.

```python
def two_pass(img, connectivity):
    img = fill_holes(img)
    labels = np.zeros_like(img, dtype=int)

    next_label = 1
    label_equivalences = {}

    # First pass
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if img[i, j] == 0:
                labels[i, j] = 0
                continue

            neighbors = []
            if i > 0 and img[i-1, j] != 0:
                neighbors.append(labels[i-1, j])
            if j > 0 and img[i, j-1] != 0:
                neighbors.append(labels[i, j-1])
            if connectivity == 8 and i > 0 and j > 0 and img[i-1, j-1] != 0:
                neighbors.append(labels[i-1, j-1])
            if connectivity == 8 and i > 0 and j < img.shape[1] - 1 and img[i-1, j+1] != 0:
                neighbors.append(labels[i-1, j+1])

            if len(neighbors) == 0:
                labels[i, j] = next_label
                label_equivalences[next_label] = next_label
                next_label += 1
            elif len(neighbors) == 1:
                labels[i, j] = neighbors[0]
            elif len(neighbors) == 2:
                labels[i, j] = min(neighbors)
                union(label_equivalences, neighbors[0], neighbors[1])
            elif len(neighbors) == 3:
                labels[i, j] = min(neighbors)
                union(label_equivalences, neighbors[0], neighbors[1])
                union(label_equivalences, neighbors[0], neighbors[2])
            elif len(neighbors) == 4:
                labels[i, j] = min(neighbors)
                union(label_equivalences, neighbors[0], neighbors[1])
                union(label_equivalences, neighbors[0], neighbors[2])
                union(label_equivalences, neighbors[0], neighbors[3])
```

- **First Pass**: The primary goal of the first pass is to assign a provisional label to each pixel and record label equivalences. The process works as follows: iterate through each pixel in the image. For each pixel, check its left, top, and top-left & top-right neighbors (the top-left & top-right neighbor is considered only for 8-connectivity).
    - If only one neighbor has a label, the current pixel adopts that label.
    - If multiple neighbors have labels, select the smallest label as the root and merge all neighboring labels under this root. The new label for the current pixel is the root label.
    - If none of the neighbors are labeled, assign a new label, which is the current maximum label + 1.

```
    # Second pass
    labe_area = {}
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if img[i, j] == 0:
                labels[i, j] = 0
                if 0 not in labe_area:
                    labe_area[0] = 0
                labe_area[0] += 1
            else:
                label = find_root(label_equivalences, labels[i, j])
                labels[i, j] = label
                if label not in labe_area:
                    labe_area[label] = 0
                labe_area[label] += 1

    # Remove small components
    for label in labe_area:
        if labe_area[label] < 600:
            labels[labels == label] = 0

    return labels
```

- **Second Pass**: The purpose of the second pass is to resolve the equivalences recorded during the first pass. Using the union-find data structure, each pixel's provisional label is replaced with its root label, ensuring consistent labeling for connected components.
- `label_area = {}`: This dictionary is used to record the area of each connected component. If a component's area is smaller than a predefined threshold (e.g., 600 pixels), it is treated as noise and removed by setting its label to 0 (black). This step helps eliminate small, irrelevant regions that might interfere with the analysis.

This approach ensures accurate labeling of connected components while effectively handling small artifacts or noise in the image.

Seed Filling Algorithm

```
"""
TODO Seed filling algorithm
"""
def seed_filling(binary_img, connectivity):
    binary_img = fill_holes(binary_img)
    height, width = binary_img.shape
    labels = np.zeros_like(binary_img, dtype=int)
    next_label = 1

    def get_neighbors(x, y):
        neighbors = []
        # 4-connectivity
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        # Add diagonal directions for 8-connectivity
        if connectivity == 8:
            directions.extend([(1, 1), (1, -1), (-1, 1), (-1, -1)])

        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if (0 <= new_x < height and 0 <= new_y < width and
                binary_img[new_x, new_y] == 255 and
                labels[new_x, new_y] == 0):
                neighbors.append((new_x, new_y))
        return neighbors

    def fill_region(start_x, start_y, label):
        stack = [(start_x, start_y)]
        while stack:
            x, y = stack.pop()
            if labels[x, y] != 0:  # 已經被標記過
                continue

            labels[x, y] = label
            neighbors = get_neighbors(x, y)
            stack.extend(neighbors)

    # 掃描整個圖像
    for i in range(height):
        for j in range(width):
            if binary_img[i, j] == 255 and labels[i, j] == 0:
                fill_region(i, j, next_label)
                next_label += 1

    # 移除小區域
    label_areas = {}
    for label in range(1, next_label):
        area = np.sum(labels == label)
        label_areas[label] = area
        if area < 600:  # 與 two_pass 使用相同的閾值
            labels[labels == label] = 0

    return labels
```

This implementation of the seed-filling algorithm labels connected components in a binary image using either 4-connectivity or 8-connectivity. Here are the key steps and functions explained:

1. **Preprocessing the Binary Image**
   - `fill_holes(binary_img)` : Preprocesses the binary image to fill small holes, ensuring better segmentation results.
   - `labels` : A zero-initialized array of the same shape as the input image to store labels.
   - `next_label` : Tracks the next available label to assign to new connected components.
2. **Neighbor Identification**
   - **Purpose**: Identifies the valid neighbors of a given pixel based on the specified connectivity.
   - **Directions**:
     - For 4-connectivity: Only horizontal and vertical neighbors.
     - For 8-connectivity: Includes diagonal neighbors as well.
   - **Conditions**:
     - Ensures neighbors are within image bounds.
     - Only considers white pixels (`binary_img[new_x, new_y] == 255`) that have not been labeled yet (`labels[new_x, new_y] == 0`).
3. **Region Filling**
   - **Purpose**: Uses a stack-based approach to label all connected pixels starting from a seed point.
   - **Process**:
     - **Initialize**: Start with the given pixel coordinates and label.
     - **Iterative Filling**: Use a stack to iteratively explore and label connected pixels.
     - **Skip Labeled Pixels**: If a pixel is already labeled, it is skipped.
4. **Labeling the Entire Image**
   - **Purpose**: Scans through the entire image to find unprocessed white pixels (`255`).
   - **Process**:
     - For each unprocessed pixel, invoke `fill_region` to label the entire connected component.
     - Increment `next_label` after processing each connected component.
5. **Removing Small Regions**
   - **Purpose**: Removes small connected components that are considered noise.
   - **Process**:
     - Calculate the area of each connected component.
     - If the area is below a predefined threshold (e.g., `600` pixels), the component is removed by setting its label to `0` .

White Patch Algorthm

```
"""
TODO White patch algorithm
"""
def white_patch_algorithm(img):
    # 分離BGR通道
    b, g, r = cv2.split(img)

    # 找出每個通道的最大值
    max_b = np.max(b)
    max_g = np.max(g)
    max_r = np.max(r)

    # 計算縮放係數（將最大值縮放到255）
    scale_b = 255.0 / max_b if max_b != 0 else 1
    scale_g = 255.0 / max_g if max_g != 0 else 1
    scale_r = 255.0 / max_r if max_r != 0 else 1

    # 調整每個通道
    b = np.clip(b * scale_b, 0, 255).astype(np.uint8)
    g = np.clip(g * scale_g, 0, 255).astype(np.uint8)
    r = np.clip(r * scale_r, 0, 255).astype(np.uint8)

    # 合併通道
    balanced_img = cv2.merge([b, g, r])
    return balanced_img
```

We first split the image into three separate channels: R, G, and B. For each channel, the brightest pixel value is assumed to represent white. Using this as a reference, we adjust the intensity of all other pixels proportionally to match the white reference.

After the adjustment, we apply `clip` to ensure that the adjusted values do not exceed 255, maintaining valid pixel intensity ranges. The adjusted values are then converted to an unsigned 8-bit integer format to ensure compatibility with standard image processing formats. Finally, the three channels are merged back together to form the corrected image.

This process effectively balances the image's color by using the brightest pixel as the white reference, which is particularly useful for correcting images with incorrect white balance.

```python
"""
TODO Gray-world algorithm
"""
def gray_world_algorithm(img):
    # 分離BGR通道
    b, g, r = cv2.split(img)

    # 計算每個通道的平均值
    avg_b = np.mean(b)
    avg_g = np.mean(g)
    avg_r = np.mean(r)

    # 計算整體平均值
    avg = (avg_b + avg_g + avg_r) / 3

    # 計算縮放係數
    scale_b = avg / avg_b if avg_b != 0 else 1
    scale_g = avg / avg_g if avg_g != 0 else 1
    scale_r = avg / avg_r if avg_r != 0 else 1

    # 調整每個通道
    b = np.clip(b * scale_b, 0, 255).astype(np.uint8)
    g = np.clip(g * scale_g, 0, 255).astype(np.uint8)
    r = np.clip(r * scale_r, 0, 255).astype(np.uint8)

    # 合併通道
    balanced_img = cv2.merge([b, g, r])
    return balanced_img
```

First, the image is split into three separate channels: R, G, and B. Then, the average intensity for each channel is calculated, along with the overall average intensity of the entire image. According to the Gray-world algorithm, the overall average intensity of an image should ideally be neutral gray. Using this assumption, the adjustment factor for each channel is calculated as `avg / avg_channel`, where `avg` is the overall average intensity, and `avg_channel` is the average intensity of the respective channel.

Each pixel's intensity is then scaled by this adjustment factor to bring the channel closer to the neutral gray assumption. This process ensures that the image's color balance is corrected while preserving the relative differences in pixel intensities. It effectively reduces color bias and produces a more natural appearance in the corrected image.

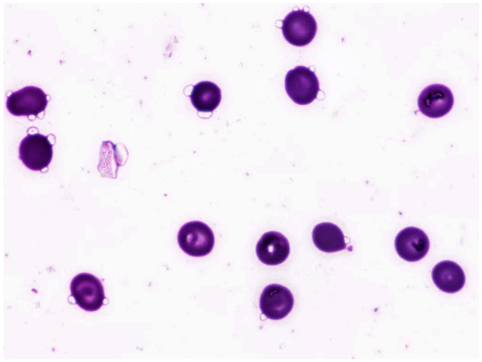
Part II. Results & Analysis

Task 1: Connected Component Analysis

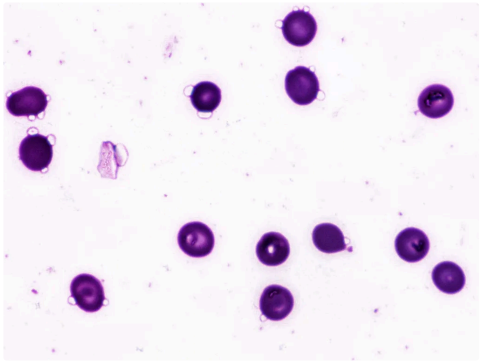
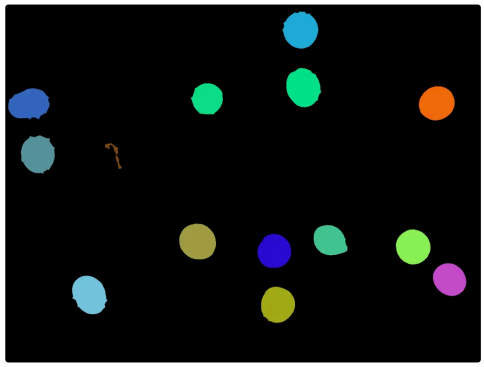
- **Two Pass Algorithm**

  Results:

  - **Input 1: 4-connectivity**




  - **Input 1: 8-connectivity**

  - **Input 2: 4-connectivity**


  - **Input 2: 8-connectivity**


The results are quite impressive, but upon closer inspection, a subtle difference is observed in **Input 1** when comparing 4-connectivity and 8-connectivity. Specifically, there is an additional small cell in the result of 8-connectivity, located near the bottom-left.

Observations:

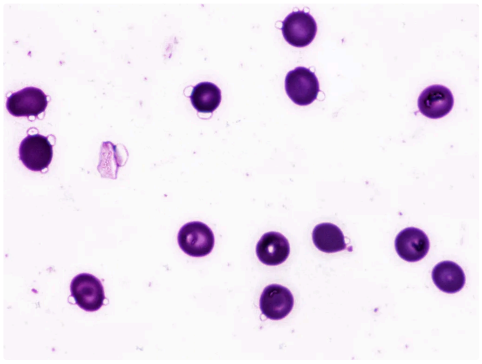
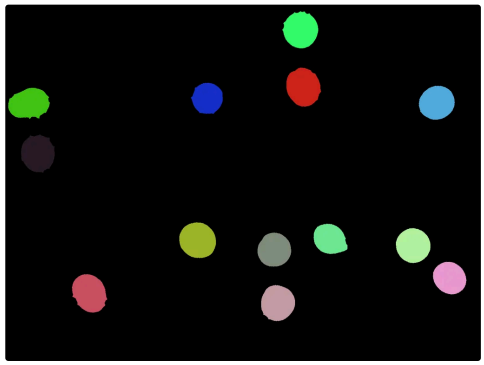
1. **Input 1: 4-connectivity vs. 8-connectivity**
    - In the 4-connectivity result, this small cell is treated as two separate regions because the pixels are only connected diagonally. Since the regions are individually too small, they are discarded as noise during the post-processing step.
    - In the 8-connectivity result, the diagonal adjacency causes these pixels to be considered part of a single connected component. This allows the cell to meet the size threshold and remain in the final segmentation.
2. **Input 2: 4-connectivity vs. 8-connectivity**
    - The results for Input 2 remain almost identical for both connectivity types. The two objects (the cats) are well-separated without any diagonal adjacency, so the choice of connectivity does not impact the segmentation.

Analysis:

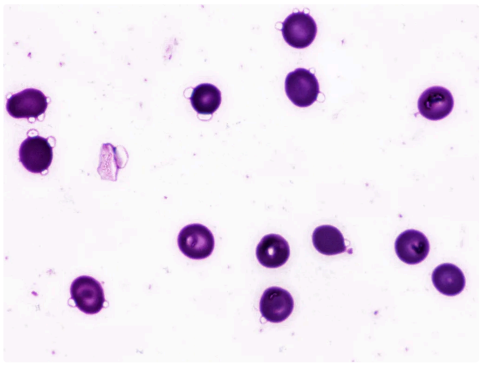
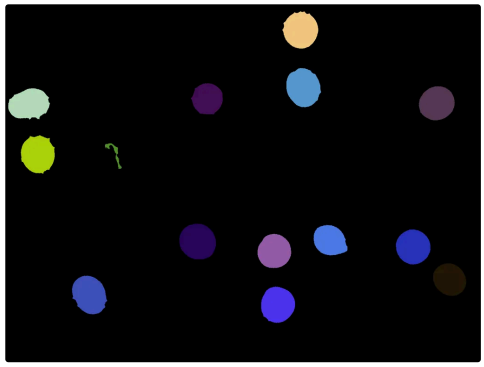
The difference in **Input 1** highlights a key distinction between 4-connectivity and 8-connectivity. While 4-connectivity treats diagonal adjacency as separate regions, 8-connectivity unites them into a single component. This behavior can lead to subtle differences in results, especially in cases with closely spaced pixels or diagonal connections.

Overall, the preprocessing steps—filling small holes and removing small connected components—were crucial in maintaining noise-free results. However, this scenario demonstrates that the choice of connectivity can have a meaningful impact on the segmentation in certain cases, depending on the spatial arrangement of the components in the image.

- **Seed Filling Algorithm**
  - **Input 1: 4-connectivity**

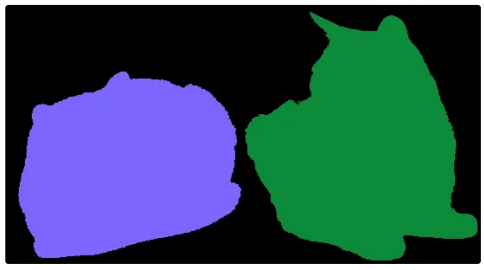



  - **Input 1: 8-connectivity**

  - **Input 2: 4-connectivity**


  - **Input 2: 8-connectivity**

The results are equally impressive, but upon closer inspection, a subtle difference is observed in **Input 1** when comparing 4-connectivity and 8-connectivity, similar to what was noted with the two-pass algorithm.

Observations:

1. **Input 1 (4-connectivity vs. 8-connectivity)**
    - In the 4-connectivity result, a small cell near the bottom-left is treated as two separate regions because the pixels are only diagonally adjacent. These smaller regions are discarded during the post-processing step due to their size.
    - In the 8-connectivity result, the diagonal adjacency unites these pixels into a single connected component. As a result, this component meets the size threshold and is retained in the final output.
2. **Input 2 (4-connectivity vs. 8-connectivity)**
    - The results for Input 2 remain nearly identical between 4-connectivity and 8-connectivity. The two objects (the cats) are well-separated without any diagonal adjacency, so the choice of connectivity does not affect the segmentation outcome.

Analysis:

Similar to the two-pass algorithm, the seed-filling algorithm demonstrates how the choice of connectivity influences the segmentation results, particularly for components with diagonal connections. In **Input 1**, 8-connectivity identifies additional regions that are missed in 4-connectivity due to the lack of diagonal linkage.

The preprocessing steps—filling small holes and removing small components—continue to ensure that noise is effectively eliminated. However, this case highlights that the choice of connectivity can result in meaningful differences when closely spaced or diagonally adjacent components are present in the image. This subtle distinction underscores the importance of considering the specific characteristics of the input image when selecting a connectivity type.

- **Comparison**

  The results from the two algorithms look very similar. This similarity can be attributed to the fact that the underlying concepts behind both algorithms are essentially the same—they both rely on examining neighboring pixels to identify connected components. The main difference lies in the order of execution:

  - The **two-pass algorithm** processes the entire image in a structured manner with two distinct passes: the first pass assigns provisional labels, and the second pass resolves label equivalences.
  - The **seed-filling algorithm** operates iteratively or recursively, expanding from a starting pixel to label an entire connected component before moving to the next.

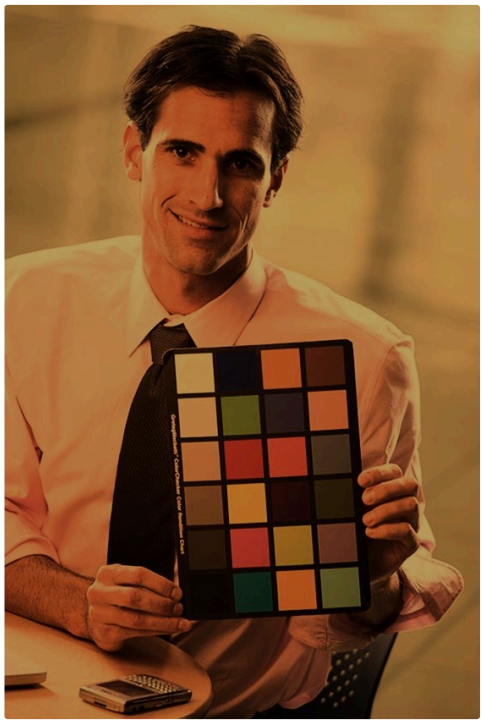
  This fundamental similarity in their approach to connectivity explains why the results are nearly identical. The choice of algorithm thus primarily depends on implementation preferences and performance considerations, rather than differences in output quality.

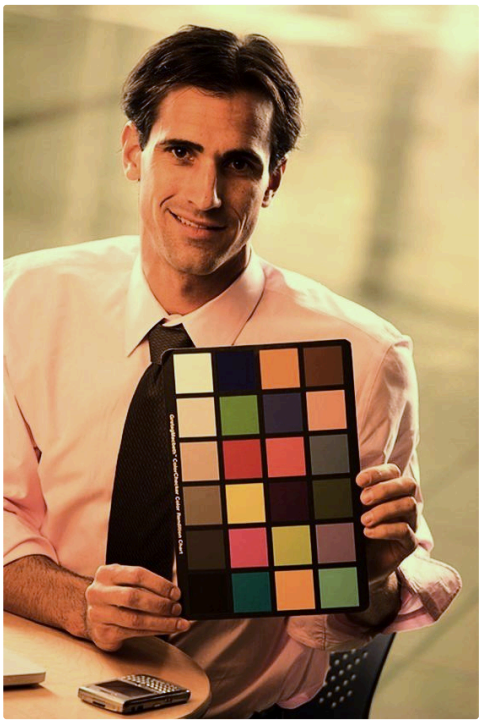
Task 2: Color Correction

- **White Patch Algorithm**

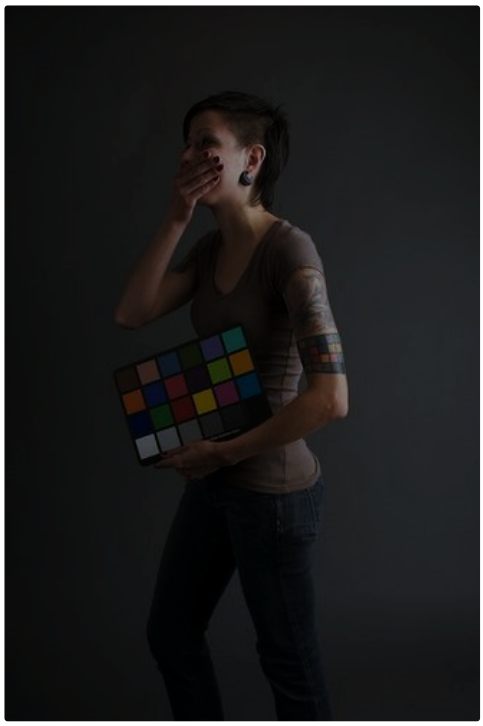
  Results:

  - **Input 1**





  - **Input 2**



  Observations:

1. **Input 1**: The result from the White Patch Algorithm shows that the strong yellowish tint in the original image has not been completely removed. While there is some improvement, the overall color still leans toward warm tones, and the white balance correction appears incomplete.

2. **Input 2**: The algorithm performs remarkably well, significantly brightening the overly dark input image. The colors in the corrected image appear natural, and the subject, as well as the ColorChecker, are much clearer and well-balanced compared to the original.

Analysis:

The results demonstrate the strengths and limitations of the White Patch Algorithm:

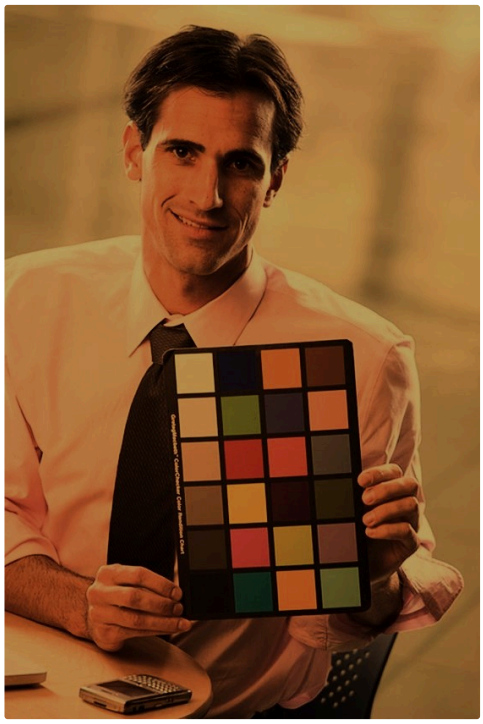
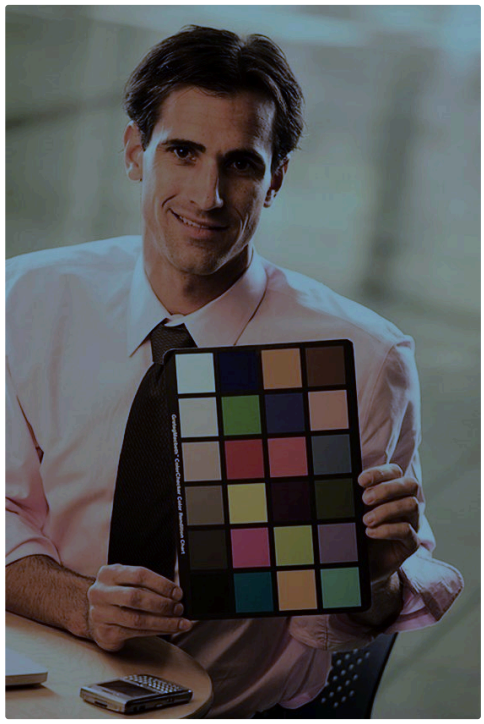
- **Input 1** highlights the algorithm's reliance on the brightest pixels to represent white. If the brightest areas in the image are not neutral white (e.g., influenced by a color cast like yellow), the algorithm may not achieve accurate white balance. This limitation results in only partial correction of the yellow tint in the image.
- **Input 2**, on the other hand, shows the algorithm's ability to adjust and balance images with sufficient dynamic range. By using the brightest pixels as a reference, it effectively brightens the image and restores color balance, even in low-light conditions.

In summary, while the White Patch Algorithm works well for correcting dark images with clear white references (as seen in Input 2), it struggles in situations where the brightest areas are influenced by a color cast, leading to less effective results, as seen in Input 1.

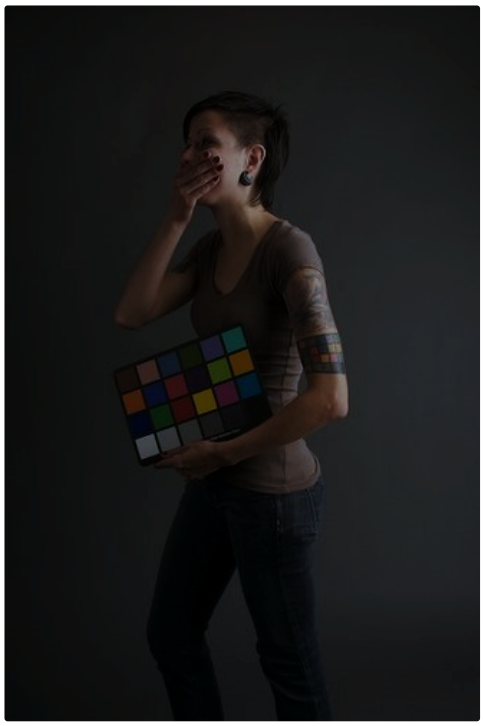
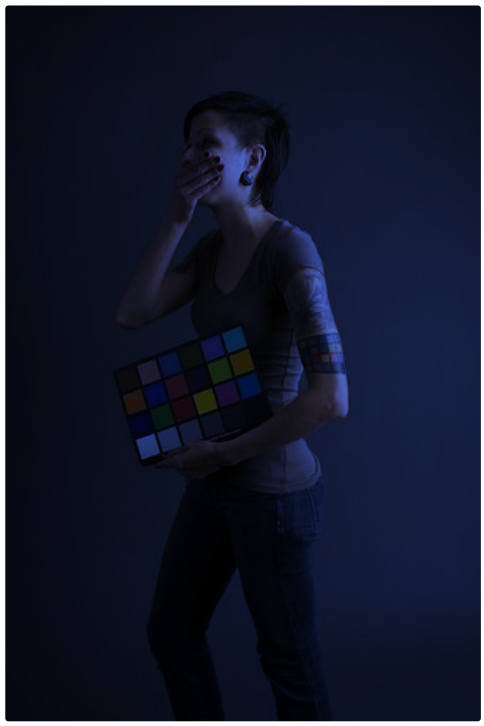
- **Gray-world Algorithm**

  Results:

  - **Input 1**

  - **Input 2**

  Observations:

1. **Input 1**: The Gray-world algorithm performs well in correcting the strong yellowish tint in the original image. The result shows a more natural white balance, with the colors appearing accurate and vibrant. The ColorChecker's colors are well restored, and the subject's skin tones look more realistic.

2. **Input 2**: The result is overly dark and has a strong blue cast. While the algorithm attempted to balance the colors based on the assumption of a neutral gray average, the lack of sufficient brightness in the input image caused an exaggerated adjustment towards blue tones, leading to an unnatural appearance.

Analysis:

The results highlight the strengths and limitations of the Gray-world algorithm:

- **Strengths**: It performs effectively in well-lit images, like Input 1, where the average color assumption aligns well with the image content. The yellow tint is successfully removed, and the overall color balance is significantly improved.
- **Limitations**: The algorithm struggles with dark images, like Input 2, where the overall average color is skewed due to insufficient lighting. This leads to overcompensation, resulting in an unnatural blue tint and darker output.

In summary, the Gray-world algorithm is suitable for images with balanced lighting and a relatively neutral color distribution. However, its reliance on the global average makes it less effective in handling low-light or high-contrast scenarios, as seen in Input 2.

- **Comparison**
  - **Input 1**
    - **White Patch Algorithm**:
      - Successfully reduces the yellow tint but does not completely eliminate it.
      - Colors are more vibrant and true-to-life compared to the Gray-world algorithm.
      - The correction is partial, leaving a slight warm tone in the image.
    - **Gray-world Algorithm**:
      - Removes the yellow tint more effectively, resulting in a more neutral white balance.
      - However, the colors appear slightly muted, lacking the vibrancy achieved by the White Patch Algorithm.
      - Overall, the correction is more consistent but less vivid.
  - **Input 2**
    - **White Patch Algorithm**:
      - Performs excellently by significantly brightening the image and restoring natural colors.
      - The white balance is well-corrected, and the subject and ColorChecker become much clearer and more realistic.
    - **Gray-world Algorithm**:
      - Struggles with this dark image, producing a result that is overly dark and tinted blue.
      - The algorithm overcompensates due to the skewed average in the input image, leading to an unnatural outcome.
  - **Summary of Comparison:**
    - **White Patch Algorithm**:
      - Strength: Excels in well-lit and dark images with clear white references, producing vibrant and natural results.
      - Limitation: Relies on the brightest pixels to be neutral white, which may lead to incomplete corrections (e.g., Input 1).
    - **Gray-world Algorithm**:
      - Strength: Handles well-lit images with balanced color distributions effectively, producing a consistent and neutral result.
      - Limitation: Struggles in low-light scenarios, leading to exaggerated color casts and darker outputs (e.g., Input 2).

Part III. Answer the questions

1. Problem Encountered and Solution:

Initially, the segmentation results were not satisfactory because the algorithm identified small artifacts and holes as separate connected components, resulting in noisy and fragmented outputs. To address this issue:

- **Solution**: I incorporated a preprocessing step using `fill_holes` to close small gaps in the image and `remove_small_regions` to eliminate connected components below a certain size threshold. These adjustments significantly improved the segmentation results by ensuring cleaner and more accurate outputs.

2. Advantages and Limitations of Two-pass and Seed-filling Algorithms:

Two-pass Algorithm:

- **Advantages**:
  - Efficient for large images due to its structured approach.
  - Resolves label equivalences systematically in two distinct passes.
  - Scalable and works well with union-find data structures.
- **Limitations**:
  - Requires two complete passes over the image, which may increase memory usage for equivalence tracking.
  - Slightly less intuitive compared to the recursive or iterative seed-filling approach.
- **Best Scenarios**: Suitable for large images with complex or numerous connected components, where performance and scalability are critical.

Seed-filling Algorithm:

- **Advantages**:
  - Simpler to implement and understand, especially for beginners.
  - Processes connected components iteratively or recursively, making it suitable for small-scale problems.
- **Limitations**:
  - Can be memory-intensive for large or complex images due to the stack or recursion depth.
  - May be slower for high-resolution images or those with numerous components.
- **Best Scenarios**: Ideal for small or moderately sized images, or when dealing with isolated connected components.

3. Advantages and Limitations of White Patch and Gray-world Algorithms:

White Patch Algorithm:

- **Advantages**:
  - Highly effective in correcting images with clear white references, producing vibrant and natural colors.
  - Simple to implement and computationally efficient.
- **Limitations**:
  - Relies on the assumption that the brightest pixel represents white, which may not hold true in all images.
  - Performs poorly when there are no clear white regions or in images with strong color casts.
- **Best Scenarios**: Suitable for well-lit images with identifiable white areas or when vibrant color correction is needed.

Gray-world Algorithm:

- **Advantages**:
  - More robust than White Patch in images without clear white regions.
  - Handles uneven lighting and a broader range of color distributions effectively.
  - Produces consistent results in natural scenes.
- **Limitations**:
  - Assumes the average color of the image is neutral gray, which may not be valid in certain scenarios.
  - May lead to muted or unnatural colors in images with skewed color distributions.
- **Best Scenarios**: Ideal for general-purpose white balance correction, particularly in evenly lit images with balanced color distributions. Performs well in scenes without strong white references.

In summary, both segmentation and white balance algorithms have their respective strengths and weaknesses, and the choice of method depends on the specific characteristics of the input image and the desired outcome.