Homework 3: Connected Component Analysis & Color Correction

Part I. Implementation:

For this homework, I implemented two main tasks: Connected Component Analysis and Color Correction. Here are the key implementation details:

Connected Component Analysis part:

```python
def to_binary(img):

    if len(img.shape) == 3:
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    else:
        gray = img

    _, binary = cv2.threshold(gray, thresh: 0, maxval: 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

    kernel = np.ones( shape: (3, 3), np.uint8)
    binary = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
    binary = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)

    binary_fill = binary.copy()
    h, w = binary.shape
    mask = np.zeros( shape: (h + 2, w + 2), np.uint8)
    cv2.floodFill(binary_fill, mask, seedPoint: (0, 0), newVal: 255)
    binary_fill = cv2.bitwise_not(binary_fill)
    binary = binary | binary_fill

    return binary // 255
```

In the to binary part. First, use Otsu's threshold to turn the picture into binary, than turn the object into white, the background into black. Second, remove the small noise and fill the small dent on the object. Last, return normalization result.

```python
def two_pass(binary_img, connectivity):

    height, width = binary_img.shape
    # First pass: assign initial labels
    labels = np.zeros( shape: (height, width), dtype=np.int32)
    current_label = 1
    parent = {}

    for i in range(height):
        for j in range(width):
            if binary_img[i, j] == 0:
                continue

            neighbors = []
            if connectivity == 4:
                if i > 0: neighbors.append(labels[i - 1, j])  # N
                if j > 0: neighbors.append(labels[i, j - 1])  # W
            else:  # 8-connectivity
                if i > 0 and j > 0: neighbors.append(labels[i - 1, j - 1])  # NW
                if i > 0: neighbors.append(labels[i - 1, j])  # N
                if i > 0 and j < width - 1: neighbors.append(labels[i - 1, j + 1])  # NE
                if j > 0: neighbors.append(labels[i, j - 1])  # W

            neighbors = [n for n in neighbors if n > 0]

            if not neighbors:  # No neighbors with labels
                labels[i, j] = current_label
                parent[current_label] = current_label
                current_label += 1
            else:  # Has labeled neighbors
                min_label = min(neighbors)
                labels[i, j] = min_label
                for n in neighbors:
                    union(parent, min_label, n)

    # Second pass: resolve equivalences
    final_labels = {}
    next_label = 1

    for i in range(height):
        for j in range(width):
            if labels[i, j] > 0:
                root = find_root(parent, labels[i, j])
                if root not in final_labels:
                    final_labels[root] = next_label
                    next_label += 1
                labels[i, j] = final_labels[root]

    return labels
```

The two-pass algorithm is implemented using a union-find data structure for efficient label equivalence management. The first pass assigns initial labels and establishes equivalence relationships, while the second pass resolves these relationships to determine the final labels.

```python
def seed_filling(binary_img, connectivity):

    height, width = binary_img.shape
    labels = np.zeros( shape: (height, width), dtype=np.int32)
    current_label = 1

    def get_neighbors(y, x):
        if connectivity == 4:
            directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        else:  # 8-connectivity
            directions = [(0, 1), (0, -1), (1, 0), (-1, 0),
                          (1, 1), (1, -1), (-1, 1), (-1, -1)]

        for dy, dx in directions:
            ny, nx = y + dy, x + dx
            if 0 <= ny < height and 0 <= nx < width:
                yield (ny, nx)

    # Process each pxl
    for i in range(height):
        for j in range(width):
            if binary_img[i, j] == 1 and labels[i, j] == 0:
                # S new region
                queue = deque([(i, j)])
                labels[i, j] = current_label

                while queue:
                    y, x = queue.popleft()
                    for ny, nx in get_neighbors(y, x):
                        if binary_img[ny, nx] == 1 and labels[ny, nx] == 0:
                            labels[ny, nx] = current_label
                            queue.append((ny, nx))

                current_label += 1

    return labels
```

The seed-filling algorithm uses a queue-based approach to iteratively process connected pixels. Starting from an unlabeled pixel, it explores all connected pixels based on the specified connectivity and assigns the same label.

Color correction part:

```python
def white_patch_algorithm(img):

    # Split image into RGB
    b, g, r = cv2.split(img)

    max_b = np.max(b)
    max_g = np.max(g)
    max_r = np.max(r)

    scale_b = 255.0 / max_b if max_b > 0 else 1.0
    scale_g = 255.0 / max_g if max_g > 0 else 1.0
    scale_r = 255.0 / max_r if max_r > 0 else 1.0

    b = np.clip(b * scale_b,  a_min: 0,  a_max: 255).astype(np.uint8)
    g = np.clip(g * scale_g,  a_min: 0,  a_max: 255).astype(np.uint8)
    r = np.clip(r * scale_r,  a_min: 0,  a_max: 255).astype(np.uint8)

    corrected_img = cv2.merge([b, g, r])

    return corrected_img
```

The white patch algorithm assumes the brightest pixel in each channel represents white and scales all pixel values accordingly to balance colors. It normalizes each channel based on its maximum value and suitable for images with well defined white points.

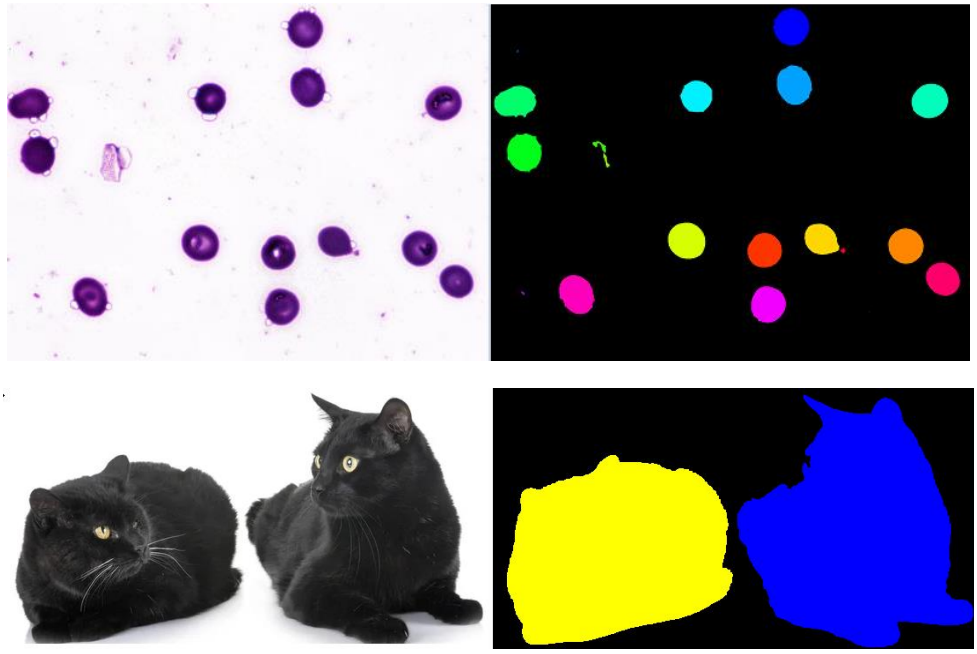
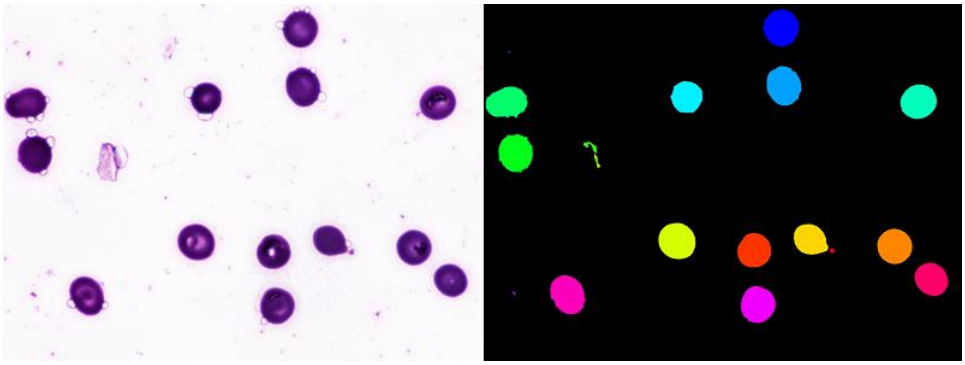
```python
def gray_world_algorithm(img):

    b, g, r = cv2.split(img)

    avg_b = np.mean(b)
    avg_g = np.mean(g)
    avg_r = np.mean(r)

    gray_avg = (avg_b + avg_g + avg_r) / 3.0

    scale_b = gray_avg / avg_b if avg_b > 0 else 1.0
    scale_g = gray_avg / avg_g if avg_g > 0 else 1.0
    scale_r = gray_avg / avg_r if avg_r > 0 else 1.0

    b = np.clip(b * scale_b,  a_min: 0,  a_max: 255).astype(np.uint8)
    g = np.clip(g * scale_g,  a_min: 0,  a_max: 255).astype(np.uint8)
    r = np.clip(r * scale_r,  a_min: 0,  a_max: 255).astype(np.uint8)

    corrected_img = cv2.merge([b, g, r])

    return corrected_img
```

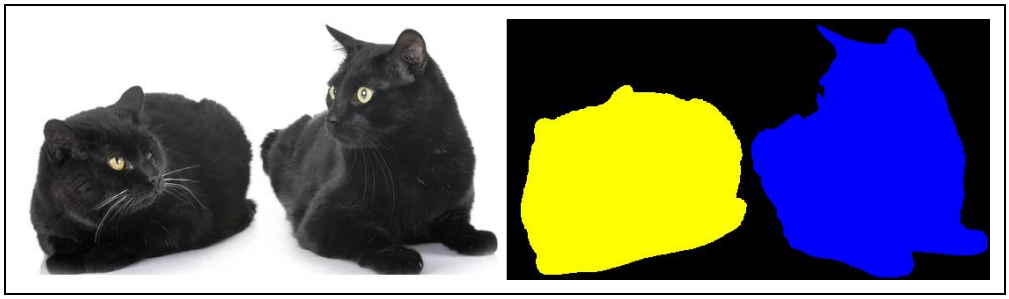
The gray-world algorithm assumes the average value of each channel should be equal. It calculates scaling factors based on the average values and applies them to normalize the colors. It adjusts each channel to achieve a balances appearance and perform well across varied scenes.

Part II. Results & Analysis:

Task 1: Connected Component Analysis

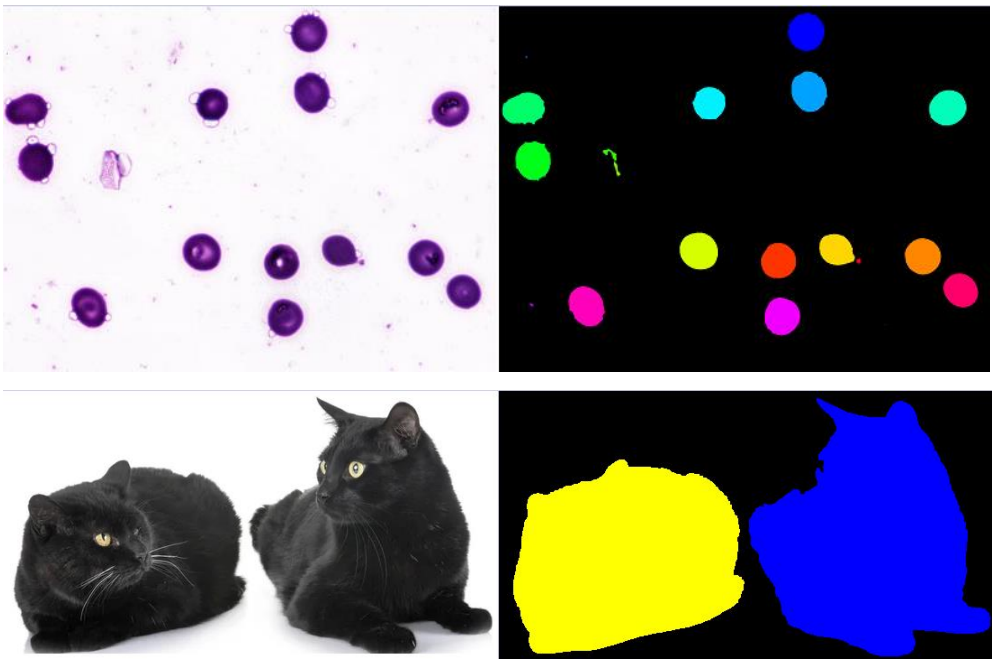
- Two-pass Algorithm:

4-connectivity

8-connectivity


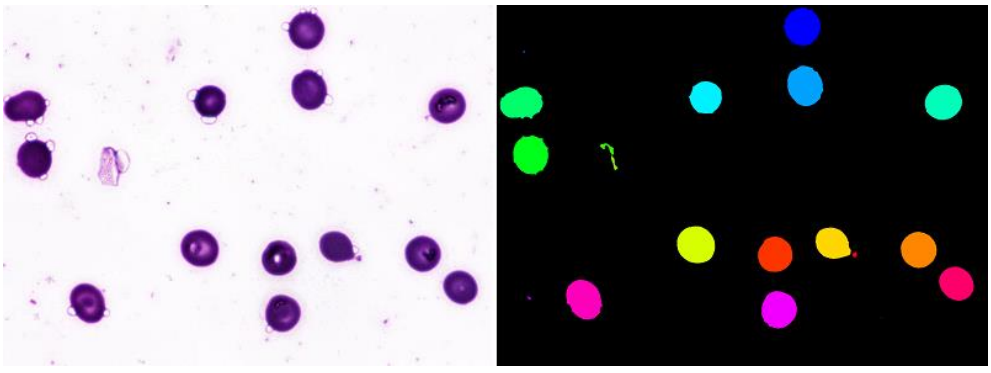


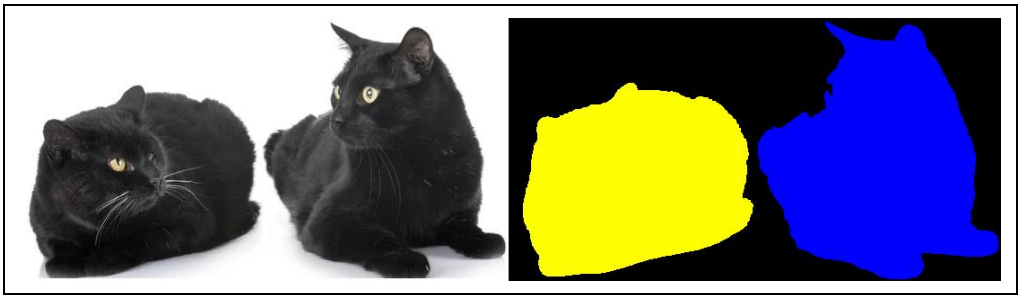
● Seed-filling Algorithm:

4-connectivity



8-connectivity





- Compare and discuss the above results:

  Honestly, if just compare the output picture between each method or between each kind of connectivity, there is literally no difference. But as far as theory is concerned, 8-connectivity can do more natural object segmentation.

  Compare between two-pass and seed-filling, Two-pass should be more robust to complex patterns cause its global approach. But in these case, picture's pattern are simple, so both method can get really good results.

Task 2: Color Correction

- White Patch Algorithm:

● Gray-world Algorithm:

● Compare and discuss the above results:

In this task, white patch algorithm can produce really bright picture, this property did well on the woman's picture, cause its original picture is really dark. But in the man's picture, it seems overcorrect the output, it cause the output a little yellow. However, it still can did good job when the original picture is in blue tint.

Another side, gray world algorithm can did well in man's photo, it makes the man's skin looks more natural than what white patch algorithm did. But in another case, cause the original picture is too dark, so the output picture do so. In total, gray world algorithm is suitable for those who want to recover photo with human, and white patch algorithm is suitable for dark photo.

Part III. Answer the questions:

1. Please describe a problem you encountered and how you solved it.
   During the implementation of the two-pass algorithm for connected component labeling, I encountered a challenge with managing label

equivalences efficiently. Initially, finding the root label for each pixel was inefficient as it required traversing through multiple parent references. To solve this issue, I implemented path compression in the find_root function where each node's parent is updated to point directly to the root during the traversal:

```
def find_root(parent, label):
    if parent[label] != label:
        parent[label] = find_root(parent, parent[label])
    return parent[label]
```

This solution significantly improved the efficiency of the second pass by reducing the time needed to resolve label equivalences, as each pixel can now find its final label with fewer operations.

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?

   For two-pass algorithm, it is suitable for complex and large photo, cause it would global view of label relationships. However, it need more memory and require more time.

   For seed-filling, its performance is decided by component size, so it may be slower for complex pattern. But it cases like these HW, cause the data picture are simple, so seed-filling can do both great and fast.

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?

   For both of these algorithms are easy to implement. But white patch algorithms require white pixel in the picture, if there’s no white pixel, the result would be yellowish. For gray-world algorithm, its cant handle strong color casts, and struggle with monochromatic scene.