Reyhane Shahrokhian 99521361

HomeWork5 of Computer Vision Course

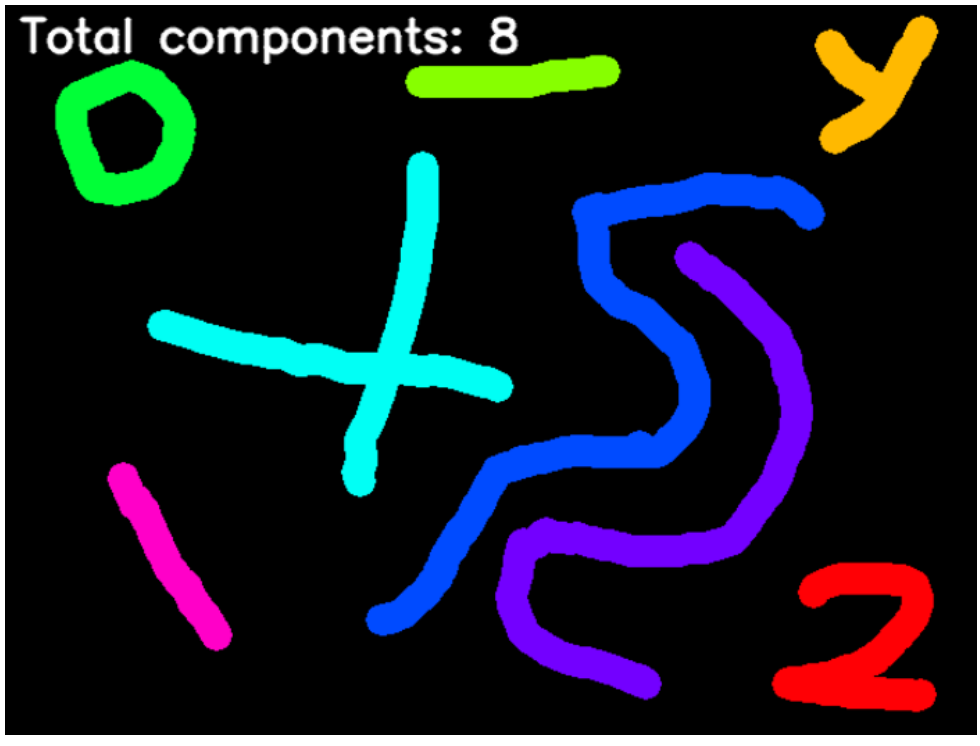
Dr. Mohammadi

Q1:

To find the connected components in an image, I implemented the find_connected_components() function, which first reads the image in grayscale. Then, we threshold the image to convert it to binary and after that by using cv2.connectedComponents(), we can find the connected components in the binary image. This function returns us the number of labels and the labeled image.

In the next step to label components with different colors, I implemented the label_components() function. This function maps the component labels to hue values for coloring and creates an HSV image where the hue corresponds to the component label. Then, it converts the HSV image to BGR and sets the background (label 0) to black.

The output:



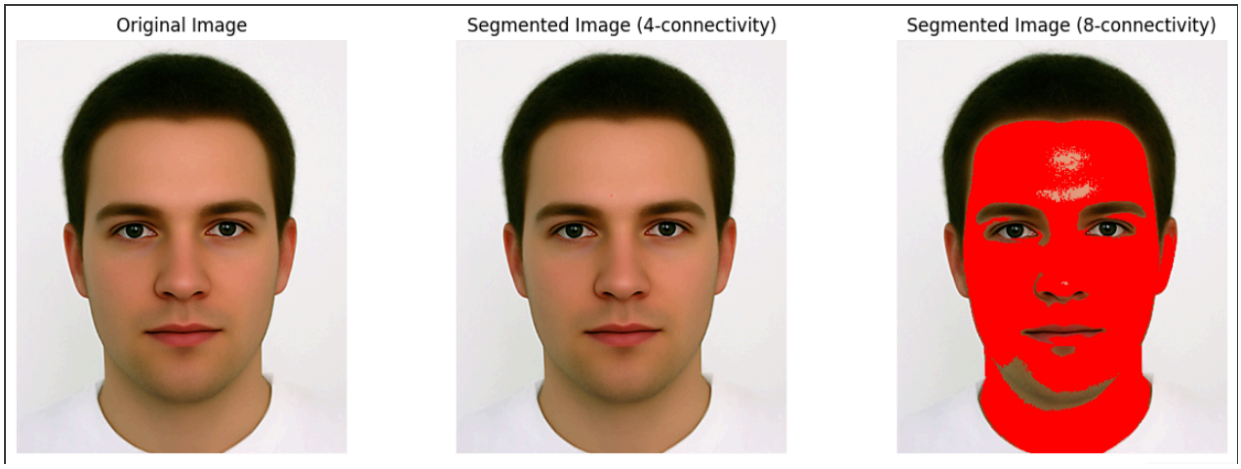
Q2:

First the image is read using OpenCV and converted to RGB since OpenCV reads images in BGR format by default.

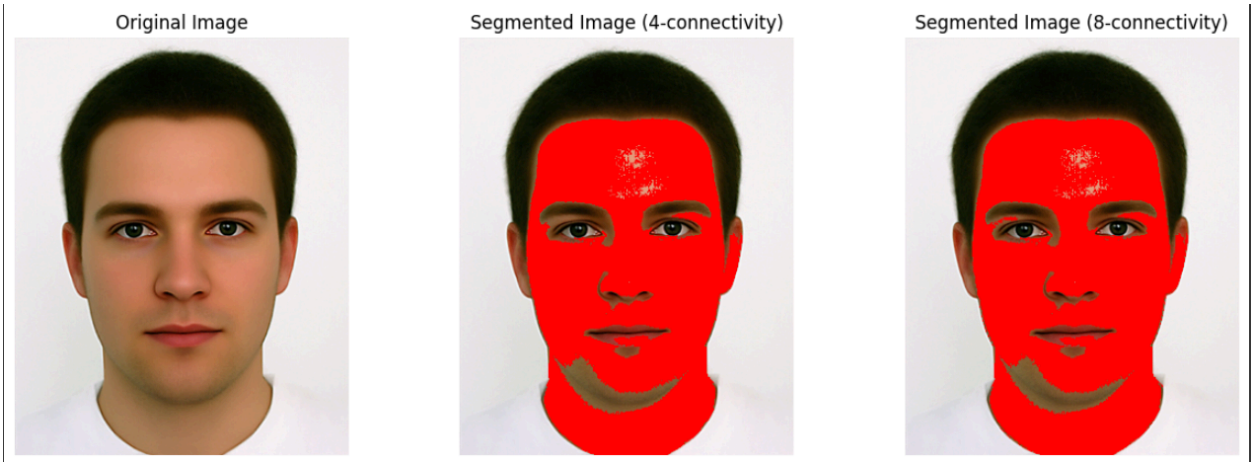
Then I implemented the get_neighbors() function which returns neighboring points based on the chosen connectivity mode (4-connectivity or 8-connectivity).

The segment() function performs the region growing algorithm by starting from the seed_point and checks all connected pixels. Then, it compares the color of each pixel with the initial seed point's color and if the color difference is within the threshold, the pixel is marked as part of the region and its color is changed. The process continues until there are no more pixels to visit.

I tested different seed points and thresholds. With many of them the 8-connectivity mode was working great but the 4-connectivity wasn't working at all. For example with seed_point= (150, 150) and threshold= 130:



But at the end with seed_point= (110, 170) and threshold= 130, I get this result:



Q3:

The Random matrix = [[3 2 9 11 10],

[5 6 12 1 4],

[15 2 14 7 6],

[13 1 5 1 8],

[9 6 3 3 10]]

Step 1: Calculate the Histogram

1: 3, 2: 2, 3: 3, 4: 1, 5: 2, 6: 3, 7: 1, 8: 1, 9: 2, 10: 2, 11: 1, 12: 1, 13: 1, 14: 1,15: 1

Step 2: Compute the probability distribution

P_1= 0.12, P_2= 0.08, P_3= 0.12, P_4= 0.04, P_5= 0.08, P_6= 0.12, P_7= 0.04, P_8= 0.04, P_9= 0.08,

P_{10}= 0.08, P_{11}= 0.04, P_{12}= 0.04, P_{13}= 0.04, P_{14}= 0.04, P_{15}= 0.04,

Step 3: Calculate the possibilities and Means of each group

- Threshold t = 6

$$w_0(6) = \sum_{i=1}^{6} P_i = 0.56$$

$$w_1(6) = 1 - 0.56 = 0.44$$

$$\mu_0(6) = \frac{\sum_{1=1}^{6} P_i i}{\sum_{1=1}^{6} P_i} = \frac{1.92}{0.56} = 3.43$$

$$\mu_1(6) = \frac{\sum_{1=7}^{15} P_i i}{\sum_{1=7}^{15} P_i} = \frac{4.72}{0.44} = 10.72$$

- Threshold t = 10

$$w_0(10) = \sum_{i=1}^{10} P_i = 0.8$$

$$w_1(10) = 1 - 0.8 = 0.2$$

$$\mu_0(10) = \frac{\sum_{1=1}^{10} P_i i}{\sum_{1=1}^{10} P_i} = \frac{4.04}{0.8} = 5.05$$

$$\mu_1(10) = \frac{\sum_{1=11}^{15} P_i i}{\sum_{1=11}^{15} P_i} = \frac{2.6}{0.2} = 13$$

Step 4: Calculate within-group Variance

- Threshold t = 6

$$\sigma_0^2(6) = \frac{\sum_{i=1}^{6} (i-\mu_0(6))^2 . P_i}{w_0(6)} = \frac{0.71+0.16+0.02+0.01+0.2+0.79}{0.56} = 3.375$$

$$\sigma_1^2(6) = \frac{\sum_{i=7}^{15} (i-\mu_1(6))^2 . P_i}{w_1(6)} = \frac{0.55+0.29+0.23+0.04+0+0.06+0.21+0.43+0.73}{0.44} = 5.77$$

$$\sigma_w^2(6) = w_0(6)\sigma_0^2(6) + w_1\sigma_1^2(6) = 0.56 \times 3.375 + 0.44 \times 5.77 = 4.428$$

- Threshold t = 10

$$\sigma_0^2(10) = \frac{\sum_{i=1}^{10}(i-\mu_0(10))^2.P_i}{w_0(10)} = \frac{1.96+0.74+0.5+0.04+0+0.11+0.15+0.35+1.25+1.96}{0.8} = 8.825$$

$$\sigma_1^2(10) = \frac{\sum_{i=11}^{15}(i-\mu_1(10))^2.P_i}{w_1(10)} = \frac{0.16+0.04+0+0.04+0.16}{0.2} = 2$$

$$\sigma_w^2(10) = w_0(6)\sigma_0^2(6) + w_1\sigma_1^2(6) = 0.8 \times 8.825 + 0.2 \times 2 = 7.46$$

As the threshold level 6 has a lower within-group variance compared to level 10, it's a better threshold.

I used this source.

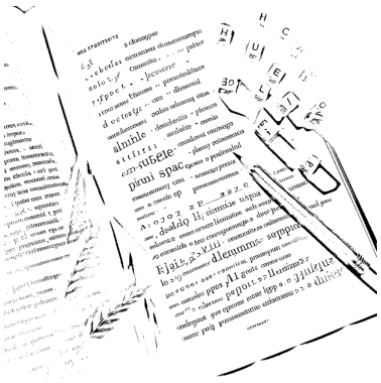
Q4:

A higher C value will result in a stricter threshold, meaning fewer pixels will be classified as foreground.

A larger block size will result in a smoother thresholding effect, but it may also be less sensitive to small details in the image.

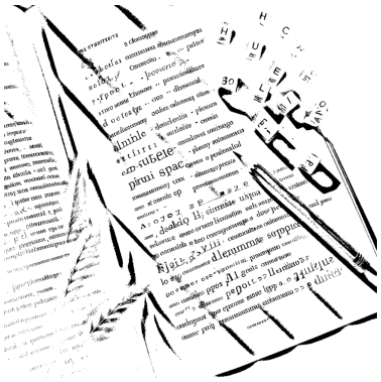
The THRESH_BINARY_INV is the opposite of THRESH_BINARY in which pixels with an intensity greater than the threshold are set to zero, and pixels below the threshold are set to the maximum value.

So accordingly now we can go through the images:

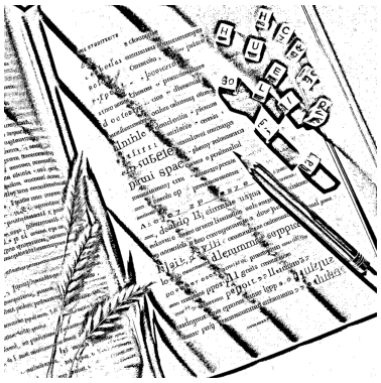


First image: As the background is white(same as the original image), the thresholdType was THRESH_BINARY. The texts at the left bottom are not clear meaning that it had the larger blocksize like 40. Also as this image is lighter than others, it had the smaller c value like 5.

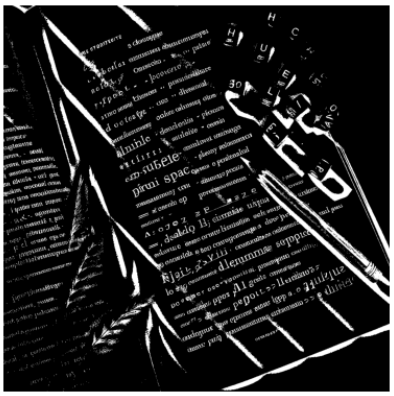
Second image: As the background is white(same as the original image), the thresholdType was THRESH_BINARY. The texts are clear at most parts meaning that it had the smaller blocksize like 21. Also as this image is darker than others, it had the larger c value like 30.



Third image: As the background is white(same as the original image), the thresholdType was THRESH_BINARY. The texts at the left bottom are not clear meaning that it had the larger blocksize like 40. Also as this image is darker than others, it had the larger c value like 30.



Fourth image: As the background is white(same as the original image), the thresholdType was THRESH_BINARY. The texts are clear at most parts meaning that it had the smaller blocksize like 21. Also as the pencil border lines aren't too pale or too bold, it had a smaller c value to not change a lot from the threshold(c = 5).



Fifth image: As the background is black(opposite of the original image), the thresholdType was THRESH_BINARY_INV. The texts at the left bottom are not clear meaning that it had the larger blocksize like 40. Also as this image is lighter than others, it had the smaller c value like 5.

Q5:

First the reflect padding is applied on the image:

22	22	22	22	33	22	22	33	22	22
22	22	22	22	33	22	22	33	22	22
22	22	33	33	33	33	33	33	22	22
22	22	22	22	33	22	33	44	22	22
22	22	22	33	44	22	33	22	22	22
22	22	22	44	22	22	44	33	22	22
33	33	22	44	22	44	33	33	22	22
33	33	33	33	33	33	22	33	22	22
33	33	33	44	33	22	44	22	44	44
33	33	33	44	33	22	44	22	44	44

The result of erosion operation:

22	22	22	22	33	22	22	33	22	22
22	22	22	22	22	22	22	22	22	22
22	22	22	22	22	22	22	22	22	22
22	22	22	22	22	33	22	22	22	22
22	22	22	22	22	22	22	22	22	22
22	22	22	22	22	22	22	22	22	22
33	22	22	22	22	22	22	22	22	22
33	22	22	22	22	22	22	22	22	22
33	33	33	33	33	22	22	22	22	44
33	33	33	44	33	22	44	22	44	44

For the dilation operation, we should first rotate the structure element 180 degree to the center and then apply it to the image:

0	0	1
0	0	1
1	1	1

22	22	22	22	33	22	22	33	22	22
22	33	33	33	33	33	33	33	33	22
22	33	33	33	33	33	44	44	44	22
22	33	33	44	44	44	44	33	22	22
22	22	44	44	44	44	44	44	33	22
22	33	44	44	44	44	44	33	33	22
33	33	44	33	44	44	33	33	33	22
33	33	44	44	44	44	44	44	44	22
33	33	44	44	44	44	44	44	44	44
33	33	33	44	33	22	44	22	44	44

Q6:

After erosion with the B1 element the orange pixels will be 1 and other pixels are 0 and after erosion of A^c with the B2 element the blue pixels will be 1 and others 0. By applying the Intersection between them, just the 2 pixels marked with red will be 1 and other 0.

This operation can find the upper right corners.

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Q7:

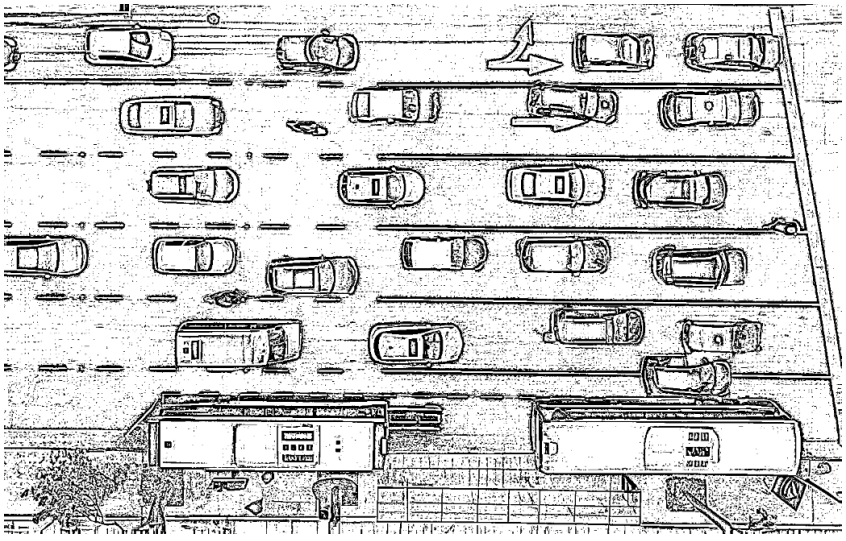
7.1:

First, after reading the image, I converted it to grayscale in order to change it to a binary image.

```
image = cv2.imread('/content/car.jpg')

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
binary_image = cv2.adaptiveThreshold(gray_image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
```

The binary image:



Then I used the morphological operation to remove noises and after that I called cv2.findcontours to find contours in the image.

```
# Delete noise
oppening = cv2.morphologyEx(binary_image, cv2.MORPH_OPEN, np.ones((8, 8), np.uint8))

# Find countours
contours, _ = cv2.findContours(oppening, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

Then by checking different values of min_countour, we can filter the wanted ones and count the number of cars.

```
min_contour = 4700
car_count = len([contour for contour in contours if cv2.contourArea(contour) > min_contour])

print(f'Number of Detected cars in the image: {car_count}')
```

```
Number of Detected cars in the image: 23
```

7.2:

First I convert the image to grayscale to apply Gaussian blur on it.

```
# Apply Gaussian blur
blurred = cv2.GaussianBlur(gray_flower, (15, 15), 0)
```

After that I apply morphological closing and opening.

```
# Structuring element for morphological operations
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (15, 15))

# Apply morphological closing
closed = cv2.morphologyEx(blurred, cv2.MORPH_CLOSE, kernel)

# Apply morphological opening
opened = cv2.morphologyEx(closed, cv2.MORPH_OPEN, kernel)
```

Then like the previous section, I called cv2.findcontours and filter contours by area to remove

noise and small detections.

```
# Find contours
contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Filter contours by area to remove noise and small detections
min_area = 9000
number_of_sunflowers = len([cnt for cnt in contours if cv2.contourArea(cnt) > min_area])

print(f"Number of sunflowers detected: {number_of_sunflowers}")
```

```
Number of sunflowers detected: 17
```

Q8:

8.1:

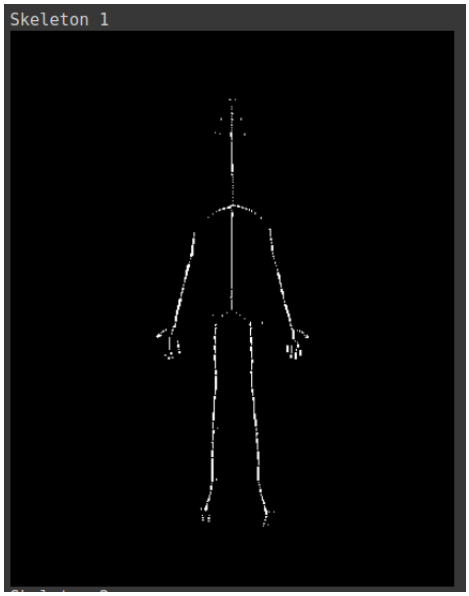
Functions explanation:

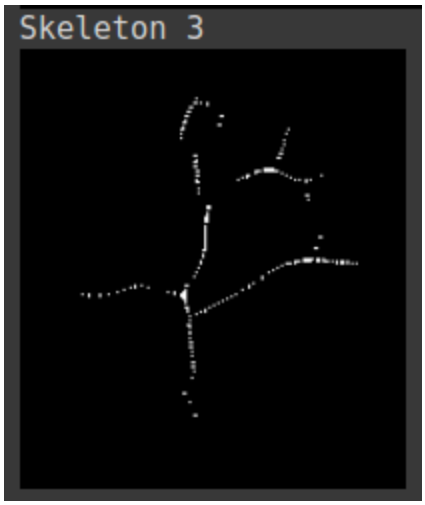
The thinning() function iteratively applies the Zhang-Suen thinning algorithm until no further changes occur. The Zhang-Suen Thinning algorithm checks the neighborhood pixels to decide if a pixel should be part of the skeleton.

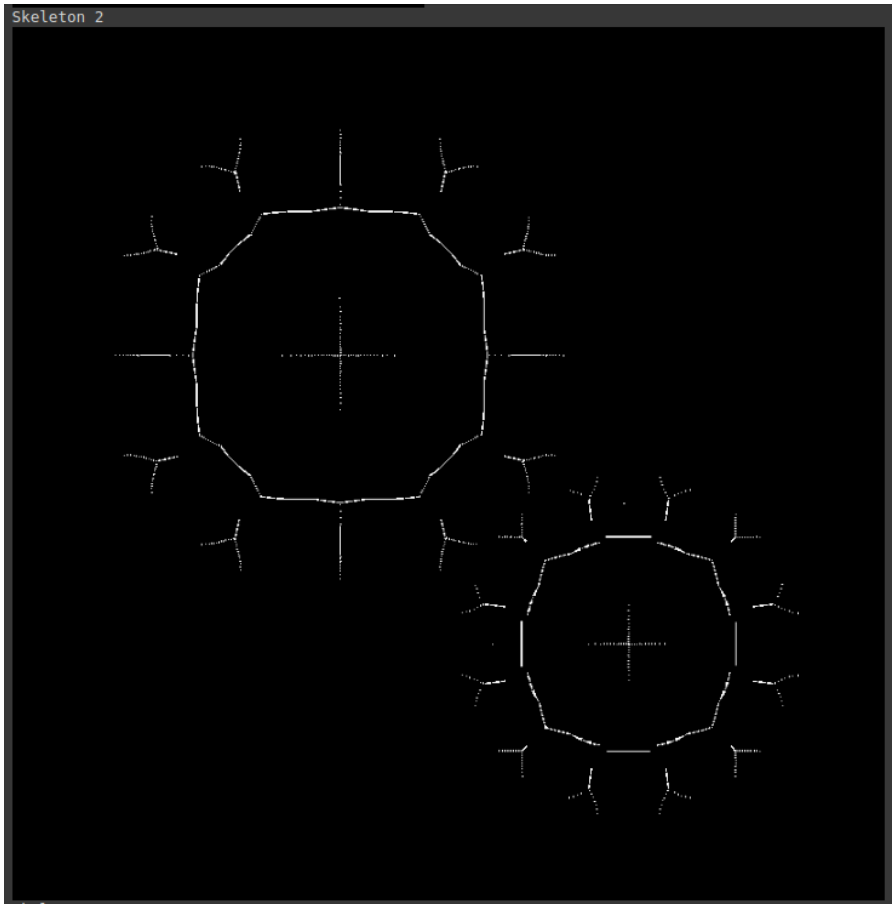
The neighborhood() function retrieves the 8-connected neighbors of a given pixel (i, j).

The erod() function reduces the white regions in the binary image by setting a pixel to white only if all pixels in the kernel area are white and the dilate() function expands the white regions by setting a pixel to white if any pixel in the kernel area is white.

The skeletonization() function, first converts the image to grayscale and then to binary using thresholding and then iteratively performs erosion, dilation, and subtracts the dilated image from the original binary image to form the skeleton. Finally, the thinning function is applied to refine the skeleton.



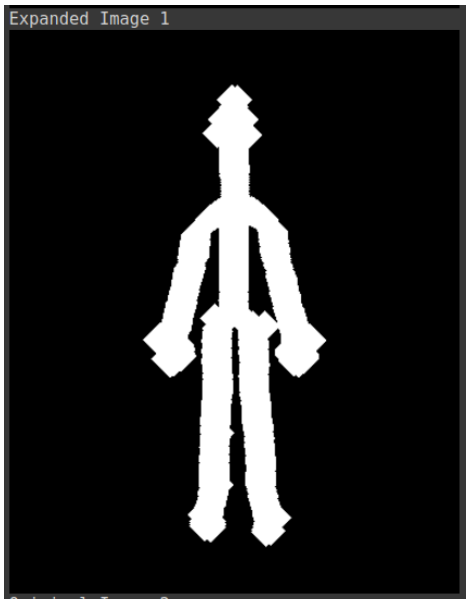


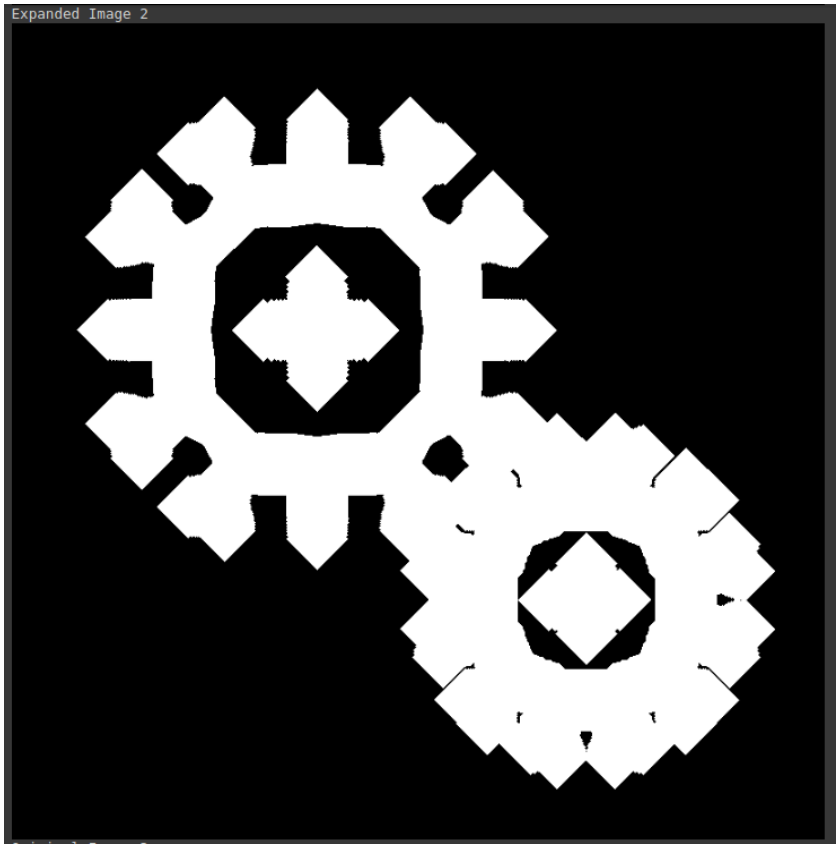


8.2:

The expand_skeleton() function expands the skeletonized image by performing dilation operations using a cross-shaped structuring element.

I used a different number of iterations for each image to get the best output.





Q9:

The structure elements are as follows:

Upper border:

0	-1	0
0	+1	0
0	0	0

lower border:

0	0	0
0	+1	0
0	-1	0

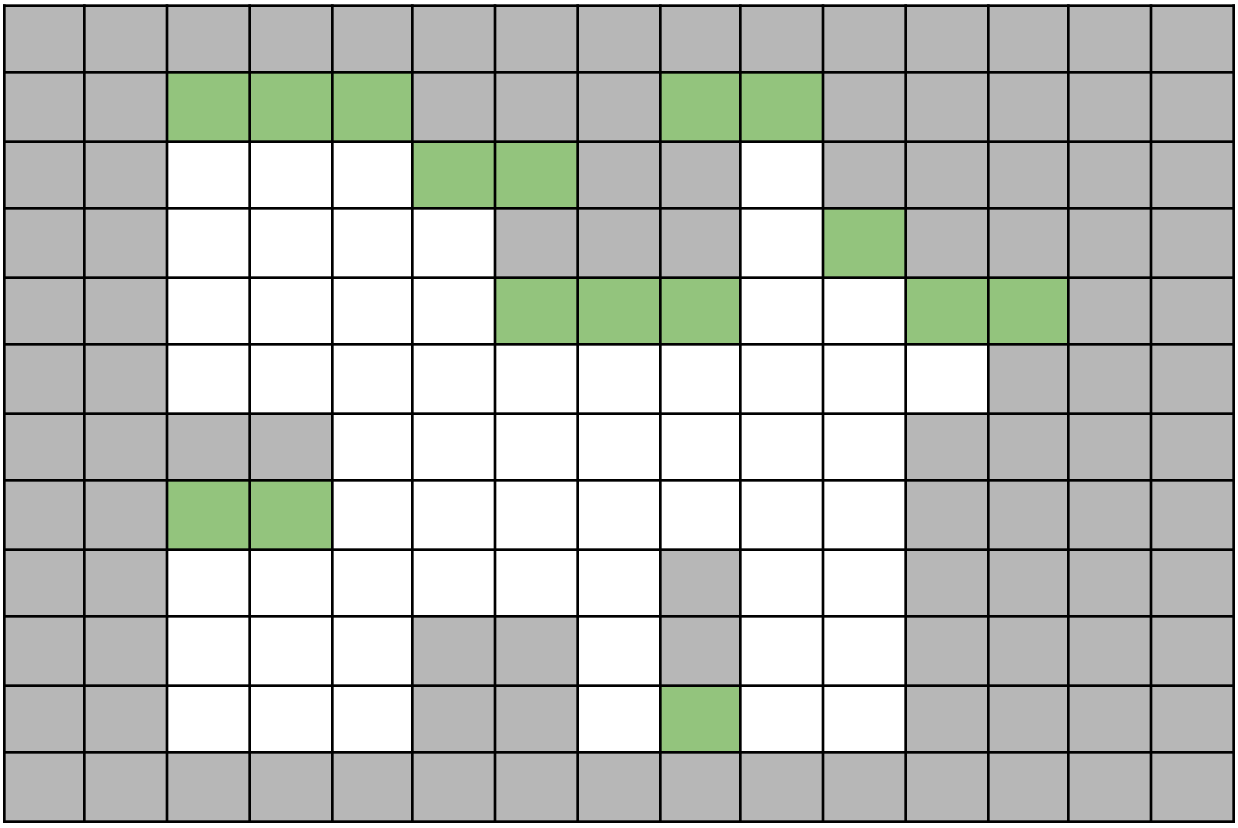
right border:

0	0	0
0	+1	-1
0	0	0

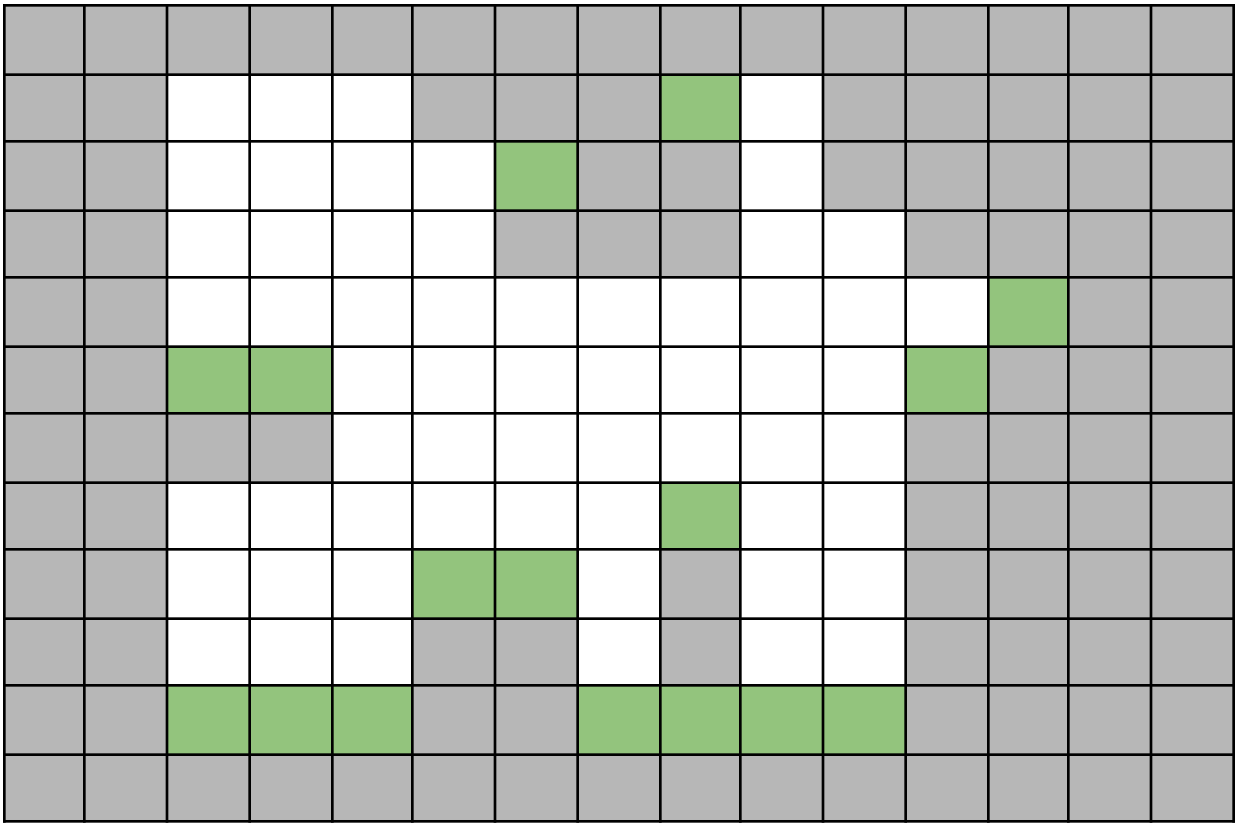
left border:

0	0	0
-1	+1	0
0	0	0

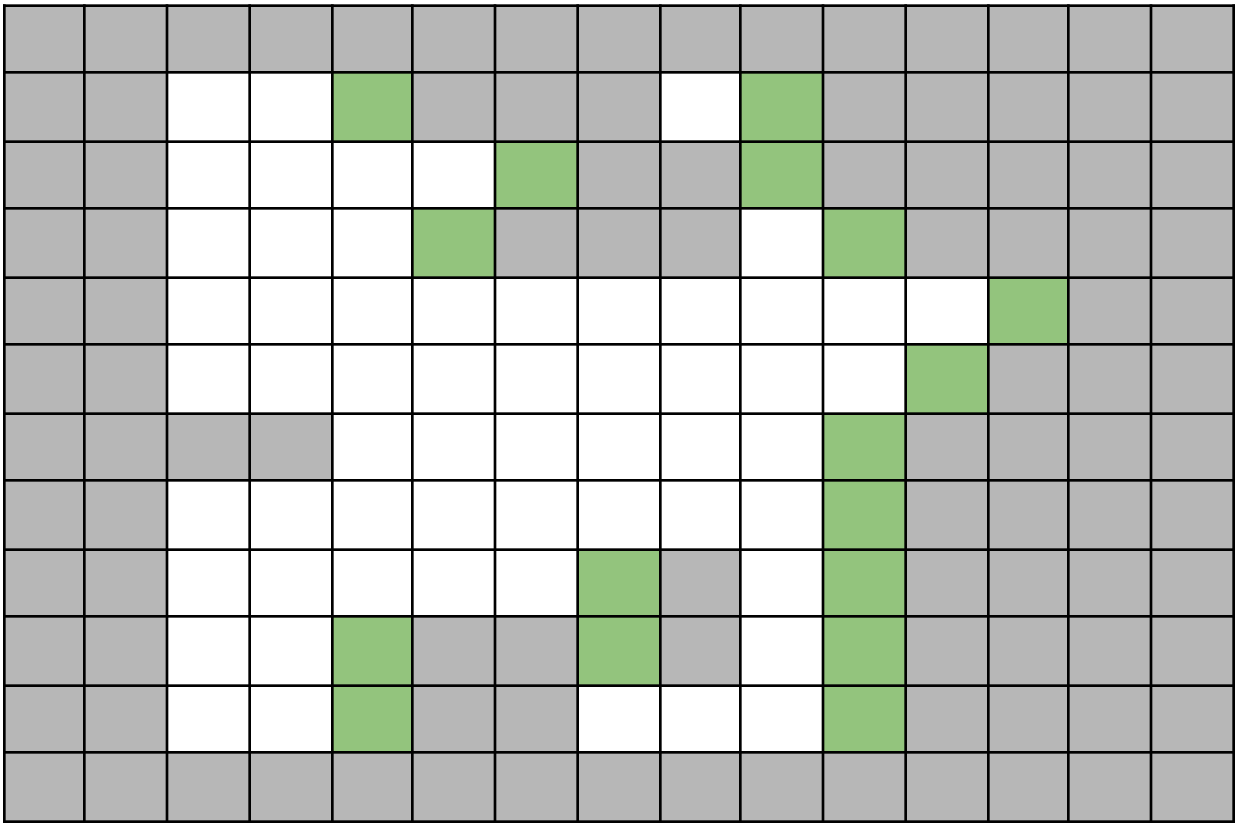
Find the upper border:



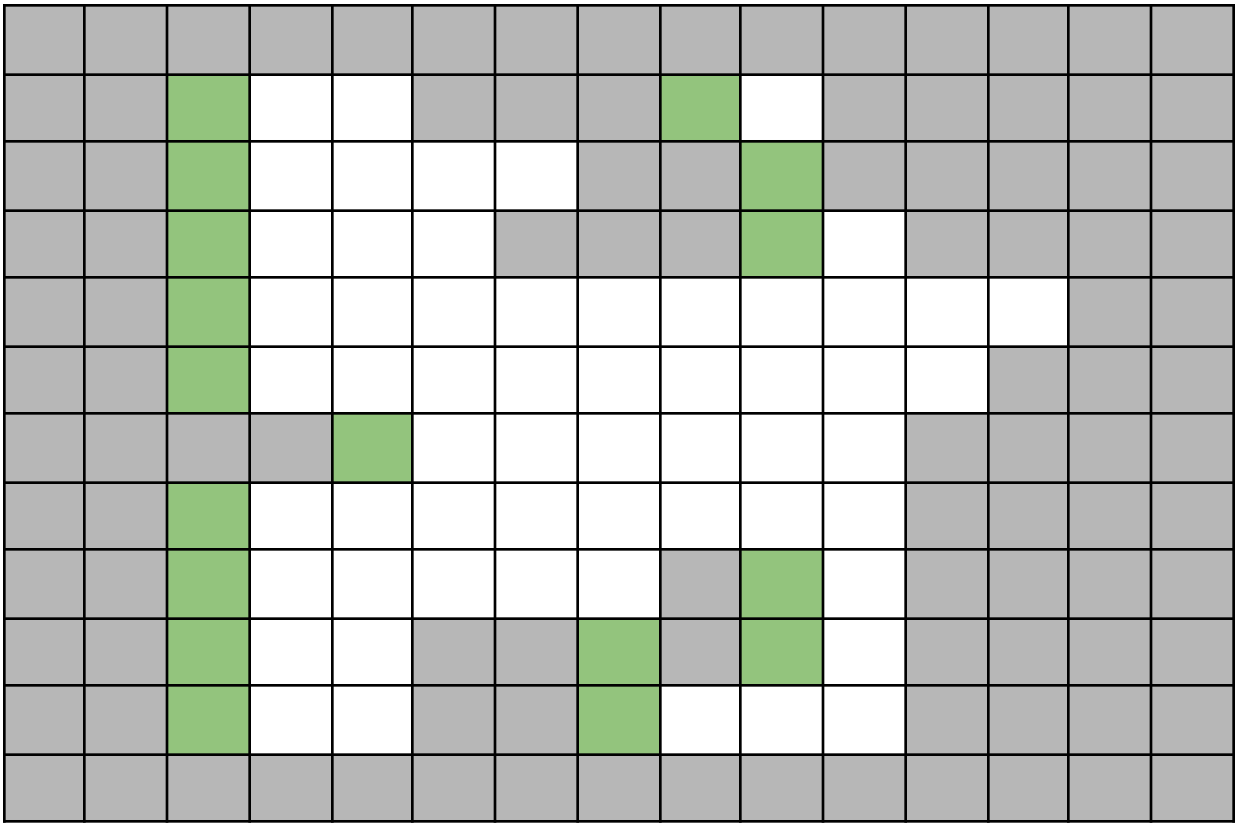
Find the lower border:



Find the right border:



Find the left border:



From the union of the above four states, the general border is obtained: