EC-312
DIGITAL IMAGE PROCESSING

ASSIGNMENT #01

HAMZA BIN SAQIB

342765

hbsaqib.ce42ceme

DE – 42 SYN – A

SYNOPSIS:

This project aims to develop an algorithm capable of decoding Braille code images into English text, with the flexibility to handle various types of images as long as they follow the basic principles of the Braille Language. The algorithm will be designed to adapt to different threshold and parameter settings to accommodate a range of input images. By successfully accomplishing this task, the algorithm will be a valuable tool for those who are blind or have low vision, enabling them to access information that would otherwise be inaccessible to them.

The first step in the process is to read in an image file in grayscale format. Grayscale images are images where each pixel is represented by a single value, typically ranging from 0 (black) to 255 (white). By converting the image to grayscale, we can simplify the image and reduce the amount of data we need to work with.

Once the image is in grayscale format, a binary threshold is applied to create a binary image. A binary image is an image where each pixel is either black (0) or white (1) based on a threshold value. In this case, the threshold is chosen to differentiate between the background and the raised Braille dots.

Next, connected component analysis is performed on the binary image to identify each raised Braille dot. Connected component analysis is a process used to group together pixels that are connected to each other. In this case, it is used to identify clusters of white pixels in the binary image, which correspond to the raised Braille dots.

After the Braille dots are identified, they are grouped into individual Braille cells. A Braille cell is a block of six raised dots arranged in two columns of three dots each, which correspond to different letters and symbols in the Braille alphabet. The script groups the raised dots together based on their proximity to each other in the image.

Finally, the Braille cells are decoded to their corresponding English characters. This is done using a set of conditions which correspond to their respective letters each.

Overall, this process allows the script to take an image of a Braille document and convert it into a digital text representation that can be read by a computer or displayed on a screen.

DESCRIPTION:

First, the necessary libraries are imported: NumPy, OpenCV (cv2), and two custom modules, Connected_Component_Analysis and Grouping.

```
import numpy as np
import cv2
import Connected_Component_Analysis
import Grouping
```

Next, the image file is read and displayed, and the maximum and minimum pixel values in the image are calculated. The threshold value is then calculated as the midpoint between the maximum and minimum values plus a constant offset.

```
# Read the Image in Grayscale Format
image = cv2.imread('Braille.png', cv2.IMREAD_GRAYSCALE)
print('\nImage Resolution:', image.shape)
# Display the Image
cv2.imshow('Braille', image)
cv2.waitKey()

# Find the Maximum & Minimum Pixel Values
max_val = np.max(image)
min_val = np.min(image)
print('Minimum Pixel Value:', min_val)
print('Maximum Pixel Value:', max_val)
# Set a Threshold Value
threshold = ((max_val - min_val) // 2) + 10
print('Threshold Value:', threshold)
```

The VSET is taken as the maximum pixel value in this case because our image is in greyscale and when it is going to be converted to binary, the components will have pixel value = 255 and the background will have pixel value = 0. Using the VSET the original image is converted to an inverted binary.

```
# Set Vset as the Maximum Value
vset = max_val
# Set a Binary Value for the Image
binary = np.zeros_like(image)
binary[image < threshold] = vset
cv2.imwrite('Braille_Binary.png', binary)
print('\n[ Binary Image Saved! ]')
```

The binary image is then saved and printed to the console.

Next, connected component analysis is performed on the binary image to identify the individual raised Braille dots. The component analysis function is provided by the Connected_Component_Analysis module.

```
# Find the Connected Components in the Binary Image
labels = Connected_Component_Analysis.connectivity(binary)
```

The **connectivity()** function takes a binary image as input and returns a label matrix with connected components labelled with distinct integers. The algorithm used to compute this matrix is called the two-pass connected component labelling algorithm. Here's a summary of how the function works:

- The function first shows the input image using the cv2.imshow() function and waits for a key event using the cv2.waitKey() function. This is simply used to display the input image for debugging purposes.
- An empty equivalency dictionary equiv_dict is defined, which will be used to store the equivalency between labels.
- A label counter label_count is initialized to 1.
- An empty label matrix labels is created with the same shape as the input image.
- The function iterates over each pixel of the input image using two nested loops. If the current pixel is a background pixel (value of 0), the loop simply continues to the next pixel. Otherwise, the function proceeds to the next step.
- A list neighbors is created to store the labels of the neighbors of the current pixel. Four types of neighbors are considered: left, top, top-left, and top-right.
- If none of the neighbors have a label assigned to them, a new label is assigned to the current pixel, and the new label is added to the equivalency dictionary as a set containing only itself. The label counter is incremented by 1.
- If there is only one neighbor with a label assigned to it, the current pixel is assigned the same label as the neighbor.
- If there are multiple neighbors with distinct labels assigned to them, the current pixel is assigned the minimum label of the neighbors. The equivalency dictionary is updated to include mappings between all pairs of distinct labels in the set of neighbors.
- The function then performs a second pass over the label matrix to replace all non-representative labels with their representative labels. The representative label for a connected component is defined as the smallest label in the set of equivalent labels for that component. The equivalency dictionary is used to determine the representative label for each connected component.
- The function returns the label matrix.

The code then initializes an image array to represent the centroid points of each dot, and proceeds to flatten the label matrix and extract the unique elements (i.e., the unique labels for each raised dot). The background label is removed, leaving only the labels for the raised dots.

```
# Initialize an Image to Represent the Centre Points
image_centroids = np.zeros_like(binary)

# Flatten the Label Matrix
labels_arr = labels.flatten()
# Extract unique elements from the Label Matrix
unique_elements = np.unique(labels_arr)
# Remove the Background Label
unique_elements = unique_elements[1:]
# Total Number of Raised Dots
print('\nNumber of Dots: ', len(unique_elements))
```

Next, the code iterates over the individual labels, identifies the centroid of each dot, and stores these centroid points in an array. The script also updates the centroid image array with the current centroid point.

```
# Array to Store the Centroids
centroids = []
# Iterate over the components and calculate their centroids
for u in unique_elements:
    component_img = np.zeros_like(binary)
    component_img[labels == u] = 255
    (y, x) = Connected_Component_Analysis.calculate_centroid(component_img)
    centroids.append((int(y), int(x)))
    image_centroids[y, x] = 255
    # print(f"Centroid of component {u}: {y, x}")

cv2.imwrite('Braille_Centroids.png', image_centroids)
print('\n[ Centroids Image Saved! ]')
cv2.imshow('Centroids', image_centroids)
cv2.waitKey()

print(f'\nNumber of Centroids: {len(centroids)} (Confirmation)')
```

The **calculate_centroid()** function takes a binary image as input and returns the centroid coordinates of the foreground (non-zero) pixels. Here's a summary of how the function works:

- The function computes the moments of the input image using the np.sum() and np.multiply() functions. The moments are computed as follows:
    - m00 = sum of all pixel intensities
    - m10 = sum of x-coordinate * pixel intensity for all foreground pixels
    - m01 = sum of y-coordinate * pixel intensity for all foreground pixels
- If m00 is zero, meaning there are no foreground pixels, the function returns None. Otherwise, the function proceeds to the next step.
- The centroid coordinates are computed as x = m10 / m00 and y = m01 / m00, and then returned as a tuple (y, x). Note that the coordinates are reversed from the traditional (x, y) order to match the row-column indexing convention used in numpy arrays.

After finding the centroids of the connected components in the binary image, the next step is to group the centroids into lines and columns, and then recognize the Braille characters.

The code uses a function called **row_wise()** from the **Grouping** module to group the centroids into lines. This function takes a list of centroids and groups them based on their y-coordinates (i.e., the row in which they appear in the image). It returns a list of lists, where each sublist contains the centroids in a single line.

```
lines = Grouping.row_wise(centroids)
```

Next, the code uses a loop to process each line. For each line, it first determines the minimum horizontal distance between centroids (**min_hor**), the minimum vertical distance between centroids (**min_ver**), the minimum diagonal distance between centroids (**min_dia**), the maximum vertical distance between centroids (**max_ver**), and the maximum diagonal distance between centroids (**max_dia**) using the **min_distance()** function from the **Grouping** module.

```
min_hor, min_ver, min_dia, max_ver, max_dia =
Grouping.min_distance(lines[0])
```

The code then loops through each group of centroids in the line, which are obtained by using the **column_wise()** function from the **Grouping** module on the centroids in the line. This function groups the centroids in each line based on their x-coordinates (i.e., the column in which they appear in the image). It returns a dictionary, where the keys are the column indices, and the values are lists of centroids in that column.

For each group of centroids in the line, the code uses the **find_char()** function from the **Grouping** module to recognize the Braille character represented by the group. This function takes the group of centroids, along with the minimum and maximum distance values calculated earlier, and returns the corresponding Braille character.

The recognized Braille characters are concatenated into a string variable called **STRING**, with spaces between characters in each line. Finally, the English translation of the Braille text is printed to the console and saved to a file called **Algorithm Output.txt**.

```
count = 0
STRING = ''
for sublist in lines:
    count += 1
    groups = Grouping.column_wise(sublist)
    groups = {k: sorted(v, key=lambda x: (x[0], x[1])) for k, v in
groups.items()}
    print(f"\nLINE: {count}")
```

```
    for index, group in enumerate(groups.values()):
        print(f"Group {index}: {group}")
        ch = Grouping.find_char(group, min_ver, min_dia, max_ver, max_dia)
        STRING += ch

    STRING += ' '

print('\nEnglish:', STRING)

with open('Algorithm Output.txt', 'w') as file:
    file.write(STRING)
```

Console:

```
Image Resolution: (3500, 2700)
Minimum Pixel Value: 10
Maximum Pixel Value: 255
Threshold Value: 132

[ Binary Image Saved! ]

Number of Dots:  1553

[ Centroids Image Saved! ]

Number of Centroids: 1553 (Confirmation)

 LINE: 1
 Group 0: [(32, 32), (32, 57)]
 Group 1: [(32, 97), (82, 97), (82, 122)]
 Group 2: [(32, 162), (57, 162), (57, 187), (82, 162)]
 Group 3: [(32, 227), (57, 227), (57, 252), (82, 227)]
 Group 4: [(32, 292), (57, 317)]
 Group 5: [(32, 357), (32, 382), (57, 382), (82, 357)]
 Group 6: [(32, 447), (57, 422), (57, 447), (82, 422)]
 Group 7: [(32, 572)]
 Group 8: [(32, 637), (32, 662), (57, 637)]
 Group 9: [(32, 702), (32, 727), (57, 702)]
 Group 10: [(32, 767)]
 Group 11: [(32, 857), (57, 832)]
 Group 12: [(32, 897), (57, 897), (57, 922), (82, 897)]
 Group 13: [(32, 987), (57, 962), (82, 962)]
.
.
.
.
 LINE: 2
 Group 0: [(232, 57), (257, 32)]
 Group 1: [(232, 97), (232, 122), (257, 122), (282, 97)]
 Group 2: [(232, 162), (232, 187)]
 Group 3: [(232, 252), (257, 227)]
 Group 4: [(232, 292), (232, 317), (257, 317)]
 Group 5: [(232, 357), (257, 382)]
 Group 6: [(232, 422), (232, 447), (257, 447), (282, 422)]
 Group 7: [(232, 512), (257, 487), (257, 512), (282, 487)]
.
.
.
.
.
```

.
.
.

```
Group 21: [(3432, 1567), (3457, 1592)]
Group 22: [(3432, 1632), (3457, 1632), (3482, 1632), (3482, 1657)]
Group 23: [(3432, 1697), (3457, 1722)]
Group 24: [(3432, 1762), (3432, 1787), (3457, 1787), (3482, 1762)]
Group 25: [(3432, 1852), (3457, 1827), (3457, 1852), (3482, 1827)]
Group 26: [(3432, 1917), (3457, 1892), (3482, 1892)]

English: currentaffairsrefertoeventsand incidentsthatarehappeninginthe presenttimeandhaveanimpacton societypoliti
```

The Complete Editable Code is Provided in a .doc file.

FLOWCHART:

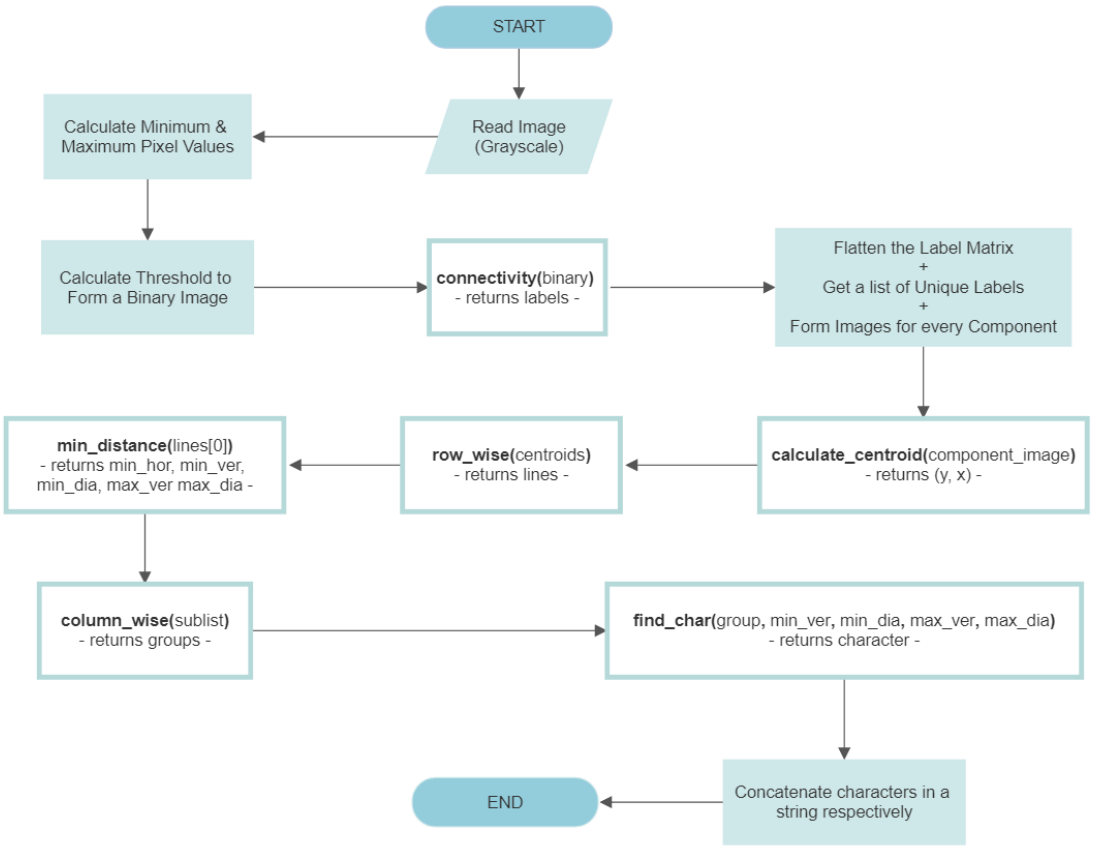
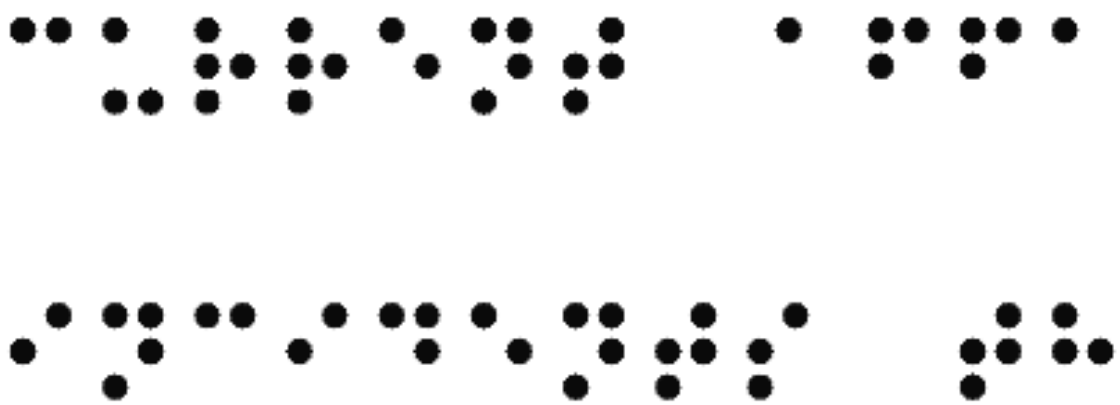
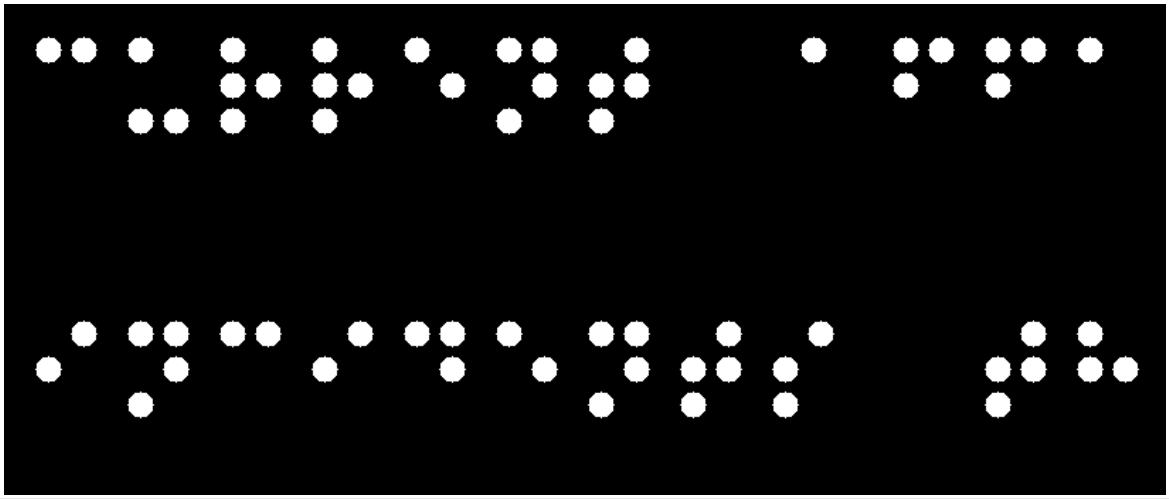


Image Segments while being processed:

Original:



Binary:



Centroid: