Logo

Description automatically generated Logo

Description automatically generated

EC-312

**DIGITAL IMAGE PROCESSING**

**ASSIGNMENT #01**

|  |
| --- |
| HAMZA BIN SAQIB |
| 342765 |
| hbsaqib.ce42ceme |

**DE** – 42 **SYN** – A

**BraileDecryption.py**

import numpy as np  
import cv2  
import Connected\_Component\_Analysis  
import Grouping  
  
*# 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)  
*# 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! ]')  
  
*# Find the Connected Components in the Binary Image*labels = Connected\_Component\_Analysis.connectivity(binary)  
  
*# 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))  
  
*# 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)')  
  
lines = Grouping.row\_wise(centroids)  
min\_hor, min\_ver, min\_dia, max\_ver, max\_dia = Grouping.min\_distance(lines[0])  
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)

**Connected\_Component\_Analysis.py**

import numpy as np  
import cv2  
  
  
def connectivity(image):  
  
 cv2.imshow('Segment', image)  
 cv2.waitKey()  
 *# Define Equivalency Dictionary* equiv\_dict = {}  
 *# Initialize the label counter* label\_count = 1  
 *# Create a Label Matrix of the same size as the Threshold Image* labels = np.zeros\_like(image, dtype=np.uint32)  
  
 *# FIRST PASS: Label the Pixels in Vset with a Temporary Label* for i in range(image.shape[0]):  
 for j in range(image.shape[1]):  
  
 if image[i][j] == 0:  
 continue  
  
 *# List of Neighbours* neighbors = []  
  
 *# Left Neighbour* if j > 0:  
 neighbors.append(labels[i][j - 1]) if labels[i][j - 1] > 0 else None  
 *# Top Neighbour* if i > 0:  
 neighbors.append(labels[i - 1][j]) if labels[i - 1][j] > 0 else None  
 *# Top Left Neighbour* if i > 0 and j > 0:  
 neighbors.append(labels[i - 1][j - 1]) if labels[i - 1][j - 1] > 0 else None  
 *# Top Right Neighbour* if i > 0 and j < image.shape[1] - 1:  
 neighbors.append(labels[i - 1][j + 1]) if labels[i - 1][j + 1] > 0 else None  
  
 *# If NO neighbors have labels, Assign a New label and Add it to the Equivalency Dictionary* if len(set(neighbors)) == 0:  
 labels[i][j] = label\_count  
 equiv\_dict[label\_count] = {label\_count}  
 label\_count += 1  
  
 *# If there is ONLY ONE Neighbor with a label, Assign that label to the Current Pixel* elif len(set(neighbors)) == 1:  
 labels[i][j] = neighbors[0]  
  
 *# If there are MULTIPLE Neighbors with Distinct labels, Assign the Minimum label to the Current Pixel* else:  
 min\_label = min(set(neighbors))  
 labels[i][j] = min\_label  
  
 *# Update the Equivalency Dictionary with the new mapping* for el in set(neighbors):  
 if el != min\_label:  
 equiv\_dict.setdefault(min\_label, set()).add(el)  
 equiv\_dict.setdefault(el, set()).add(min\_label)  
  
 *# SECOND PASS: Re-Label the Connected Components using the Equivalency Dictionary* unique\_labels = np.unique(labels)  
 for label in unique\_labels[1:]:  
 *# Find the Representative Label for this Connected Component* rep = label  
 while True:  
 if min(equiv\_dict[rep]) < rep:  
 rep = min(equiv\_dict[rep])  
 else:  
 break  
  
 *# Assign the Representative Label to ALL Pixels in the Connected Component* labels[labels == label] = rep  
  
 return labels  
  
  
def calculate\_centroid(img):  
 *# Calculate the Moments of the Image* m00 = np.sum(img)  
 m10 = np.sum(np.multiply(img, np.arange(img.shape[1]).reshape((1, -1))))  
 m01 = np.sum(np.multiply(img, np.arange(img.shape[0]).reshape((-1, 1))))  
  
 *# Calculate the Centroid Coordinates* if m00 != 0:  
 x = int(m10 / m00)  
 y = int(m01 / m00)  
 return y, x  
 else:  
 return None

**Grouping.py**

from math import dist  
import math  
  
  
def min\_distance(arr):  
 *"""Calculate Minimum - Vertical / Horizontal / Diagonal - Distances between Pixels"""* min\_horizontal = float('inf')  
 min\_vertical = float('inf')  
 min\_diagonal = float('inf')  
  
 for i in range(len(arr)):  
 for j in range(i + 1, len(arr)):  
 *# Horizontal distance* if arr[i][0] == arr[j][0]:  
 distance = abs(arr[i][1] - arr[j][1])  
 min\_horizontal = min(min\_horizontal, distance)  
  
 *# Vertical distance* elif arr[i][1] == arr[j][1]:  
 distance = abs(arr[i][0] - arr[j][0])  
 min\_vertical = min(min\_vertical, distance)  
  
 *# Diagonal distance* elif abs(arr[i][0] - arr[j][0]) == abs(arr[i][1] - arr[j][1]):  
 distance = dist(arr[i], arr[j])  
 min\_diagonal = min(min\_diagonal, distance)  
  
 max\_vertical = min\_vertical \* 2  
 max\_diagonal = math.sqrt(max\_vertical \*\* 2 + min\_vertical \*\* 2)  
 return min\_horizontal, min\_vertical, min\_diagonal, max\_vertical, max\_diagonal  
  
  
def row\_wise(arr: list[tuple[int, int]]):  
 *"""Grouping the Dots for Every 3 Rows (Makes a Line of Braille Code)"""* lines = []  
 count = 1  
 prev = arr[0][0]  
 start = 0  
 for i in range(len(arr)):  
 if arr[i][0] > prev:  
 count += 1  
 prev = arr[i][0]  
 if count > 3:  
 group = arr[start:i]  
 lines.append(group)  
 count = 1  
 start = i  
 prev = arr[i][0]  
 if i == len(arr) - 1:  
 group = arr[start:i+1]  
 lines.append(group)  
  
 return lines  
  
  
def column\_wise(arr: list[tuple[int, int]]):  
 *"""Grouping the Lines for Every 2 Adjacent Columns (Makes a Character of Braille Code)"""* min\_hor, min\_ver, min\_dia, max\_ver, max\_dia = min\_distance(arr)  
 groups = {}  
 *# Compare ALL Tuples* for i in range(len(arr)):  
 for j in range(i + 1, len(arr)):  
 *# Chessboard Distance Between 2 points* distance = math.sqrt((arr[j][0] - arr[i][0]) \*\* 2 + (arr[j][1] - arr[i][1]) \*\* 2)  
 if i not in groups and arr[i] != (0, 0):  
 groups[i] = set()  
 groups[i].add(arr[i])  
 *# If Distance Satisfies the Criteria to be in the same 3x2 Matrix* if distance == min\_ver or distance == max\_ver or distance == min\_dia or distance == max\_dia:  
 if j not in groups[i]:  
 groups[i].add(arr[j])  
 arr[j] = (0, 0)  
  
 return groups  
  
  
def find\_char(s: set, min\_ver: float, min\_dia: float, max\_ver: float, max\_dia: float):  
 *"""Find the English Alphabet Corresponding to Each Group"""* if len(s) == 1:  
 return 'a'  
  
 if len(s) == 2:  
 sec = s.pop()  
 fir = s.pop()  
 if dist(fir, sec) == min\_ver:  
 return 'c' if fir[0] == sec[0] else 'b'  
 if dist(fir, sec) == min\_dia:  
 return 'e' if (fir[0] > sec[0] and fir[1] > sec[1]) or (fir[0] < sec[0] and fir[1] < sec[1]) else 'i'  
  
 return 'k'  
  
 if len(s) == 3:  
 thi = s.pop()  
 sec = s.pop()  
 fir = s.pop()  
 if fir[1] == sec[1] == thi[1]:  
 return 'l'  
 if dist(fir, sec) == min\_ver and dist(fir, thi) == min\_dia:  
 if sec[1] == thi[1]:  
 return 'd'  
 elif fir[1] == sec[1]:  
 return 'h'  
 if dist(fir, sec) == dist(fir, thi) == min\_ver:  
 return 'f'  
 if dist(fir, sec) == min\_dia and dist(fir, thi) == min\_ver:  
 return 'j'  
 if dist(fir, sec) == min\_ver and dist(fir, thi) == max\_ver:  
 return 'm'  
 if dist(fir, sec) == min\_dia and dist(fir, thi) == max\_ver:  
 return 'o'  
 if dist(fir, sec) == min\_dia and dist(fir, thi) == max\_dia:  
 return 's'  
 if dist(fir, sec) == max\_ver and dist(fir, thi) == max\_dia:  
 return 'u'  
  
 if len(s) == 4:  
 fou = s.pop()  
 thi = s.pop()  
 sec = s.pop()  
 fir = s.pop()  
 if fir[1] == thi[1] and sec[1] == fou[1]:  
 if dist(fir, thi) == dist(sec, fou) == min\_ver:  
 if sec[0] == thi[0]:  
 return 't'  
 else:  
 return 'g'  
 elif dist(fir, thi) == dist(sec, fou) == max\_ver:  
 return 'x'  
 if fir[1] == fou[1] and sec[1] == thi[1] and dist(fir, sec) == min\_ver and dist(thi, fou) == min\_dia:  
 return 'n'  
 if fir[1] == thi[1] == fou[1]:  
 if dist(fir, sec) == min\_ver:  
 return 'p'  
 elif dist(fir, sec) == min\_dia:  
 return 'w'  
 if fir[1] == sec[1] == fou[1]:  
 return 'r'  
 if fir[1] == sec[1] == thi[1]:  
 return 'v'  
  
 return 'z'  
  
 if len(s) == 5:  
 fif = s.pop()  
 fou = s.pop()  
 thi = s.pop()  
 sec = s.pop()  
 fir = s.pop()  
 if sec[1] == thi[1] == fif[1]:  
 return 'y'  
 elif fir[1] == thi[1] == fif[1]:  
 return 'q'  
  
 return '-'