COMS W4735: Assignment 3

Geneva Ng
11 April 2023

Step 1: Raw Data

Below is my table of outputs that contains all the relevant information pertaining to each building. Below it, you will see the code that generated this output. I've included comments that explain each algorithm and threshold.

intensity	name	(x,y) of CoM	pixel area	L/R corners	MBR Diag	Intersectors
255	Carman	[38, 479]	1449	[(73, 490), (4, 469)]	72	[]
246	ButlerLibrary	[132, 460]	5546	[(179, 490), (85, 431)]	110	[]
236	Lerner	[38, 446]	2828	[(72, 467), (4, 426)]	79	[]
217	Hamilton&Hartley&Wallach&JohnJay	[240, 417]	12008	[(270, 490), (191, 338)]	171	[]
208	Journalism&Furnald	[30, 363]	5852	[(81, 414), (4, 338)]	108	[]
198	CollegeWalk	[137, 322]	4658	[(274, 331), (0, 314)]	274	[]
189	Kent	[233, 300]	1560	[(272, 310), (194, 290)]	80	[]
179	Dodge	[41, 301]	1694	[(80, 311), (3, 289)]	80	[]
170	AlmaMater	[136, 276]	196	[(143, 283), (129, 269)]	19	[]
161	Buell	[208, 253]	360	[(220, 261), (196, 246)]	28	[]
151	Philosophy	[258, 263]	1242	[(272, 286), (245, 240)]	53	[]
142	Lewisohn	[17, 259]	1456	[(31, 285), (3, 233)]	59	[EarlHall]
255	EarlHall	[49, 221]	814	[(68, 233), (31, 211)]	43	[Lewisohn]
123	StPaulChapel	[226, 222]	1200	[(251, 234), (201, 210)]	55	[]
113	LowLibrary	[135, 221]	4692	[(169, 256), (101, 187)]	96	[]
104	Mathematics	[17, 182]	1344	[(31, 206), (3, 158)]	55	[]
94	Fayerweather	[259, 176]	1250	[(272, 201), (247, 151)]	55	[]
85	Avery	[204, 175]	1200	[(215, 201), (191, 151)]	55	[]
76	OldComputerCenter	[96, 136]	286	[(103, 147), (90, 125)]	25	[]
66	Chandler&Havemeyer	[37, 119]	5082	[(80, 147), (3, 81)]	101	[]
57	Schermerhorn	[233, 120]	6440	[(273, 147), (181, 77)]	115	[Mudd&EngTerrace&Fairchild&CS]
47	Uris	[142, 99]	6434	[(175, 146), (110, 47)]	118	[Mudd&EngTerrace&Fairchild&CS]
38	NorthwestCorner	[16, 40]	1898	[(29, 77), (3, 4)]	77	[]
255	Mudd&EngTerrace&Fairchild&CS	[224, 35]	8798	[(272, 86), (166, 3)]	134	[Schermerhorn, Uris]
19	SchapiroCEPSR	[143, 20]	1360	[(163, 37), (123, 3)]	52	[]
9	Pupin	[76, 14]	1824	[(115, 27), (39, 3)]	79	[]

```
import cv2
import numpy as np
import pandas as pd

'''
This first section of code is where I set up everything I'm going to be
using later on, like a helper functions and the dictionary
that holds each bulding name as its key and all the metrics we need on every
building as an element in the value array for that key. This is also where I read in
building names off the Table.txt file, store them in a list, reverse it, then use
that to name each contour as the for-loop you'll see further down reads them from "Labled.pgm".

HELPER FUNCTION FOR BOX OVERLAP EXPLANATION: takes two arguments rect1 and rect2 in the format
[(x1, y1), (x2, y2)] where (x1, y1) and (x2, y2) are the coordinates of the upper right and lower left
```

```python
corners of a rectangle. The function checks if the two rectangles overlap with each other by comparing the
x-coordinates and y-coordinates of the rectangles, and returns True if they do and False if they don't.
'''

#HELPER FUNCTION FOR BOX OVERLAP
def check_overlap(rect1, rect2):

    rect1_ur = rect1[0]
    rect1_ll = rect1[1]
    rect2_ur = rect2[0]
    rect2_ll = rect2[1]

    # Check for overlap
    if rect1_ur[0] < rect2_ll[0] or rect2_ur[0] < rect1_ll[0]:
        return False # Rectangles don't overlap horizontally
    if rect1_ur[1] < rect2_ll[1] or rect2_ur[1] < rect1_ll[1]:
        return False # Rectangles don't overlap vertically
    return True

# INITIALIZE DICTIONARY
grand_dict = {}

# READ IN TABLE.TXT TO POPULATE NAMELIST
namelist = []
with open("Table.txt") as f:
    lines = f.readlines()
for line in lines:
    items = line.split() # Split the line by whitespace
    name = items[1] # Store the second item as a string labeled "name"
    namelist.append(name)
namelist = namelist[::-1] # Reverse namelist to match contour read-in order in the next block
i = 0

# READ IMAGE + LOCATE SHAPES
image = cv2.imread('Labeled.pgm', cv2.IMREAD_COLOR)
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
contours, _ = cv2.findContours(gray_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

'''
This is where I begin to do the actual work of fleshing out the metrics for each building.
For each metric, it's calculated then added to the end of the value array for each key. By the
end of this code, you have a dictionary with every building and their respective list of traits.
'''

# ADD SHAPES TO DICTIONARY
for i, contour in enumerate(contours):

    # FIND OBJECT
    x, y, w, h = cv2.boundingRect(contour)
    obj = image[y:y+h, x:x+w]

    # CALCULATE OBJECT INTENSITY
    intensity = np.max(obj)

    #ADD TO GRAND_DICTIONARY
    obj_name = namelist[i]
    i += 1
    grand_dict[obj_name] = [intensity]
    #ADD CENTER OF MASS TO GRAND DICT VALUE
    M = cv2.moments(contour)
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])
    com = [cx, cy]
    grand_dict[obj_name].append(com)

    #ADD TOTAL PIX AREA TO GRAND DICT VALUE
    rect = cv2.minAreaRect(contour)
    box = cv2.boxPoints(rect)
    box = np.intp(box)
    area = int(rect[1][0] * rect[1][1])
    grand_dict[obj_name].append(area)

    #IMAGE TESTER LINE - GENERATE PIC WITH MBR BOXES
    cv2.drawContours(image, [box], 0, (255, 0, 0), 2)

    #ADD L/R CORNERS + DIAG OF BOX TO GRAND DICT
    x_coords = [box[i][0] for i in range(4)]
    y_coords = [box[i][1] for i in range(4)]
    x_min = min(x_coords)
    y_min = min(y_coords)
    x_max = max(x_coords)
    y_max = max(y_coords)
    diag = int(np.sqrt((x_max - x_min)**2 + (y_max - y_min)**2))
    lowLeft = (x_min, y_min)
    upRight = (x_max, y_max)
    corners = [upRight, lowLeft]
    grand_dict[obj_name].append(corners)
    grand_dict[obj_name].append(diag)

    # DRAW BOUNDING BOX FOR THE SPECIFIED KEY
    cv2.rectangle(image, (x, y), (x+w, y+h), (0, 0, 255), 2)

'''
This block iterates through the grand_dict and uses the "check_overlap" helper function
to create an array of possible neighbors for each building. It then adds this array to the end
of each building's trait array in grand_dict.
'''

# CREATE NEIGHBORS ARRAY FOR EACH KEY IN GRAND_DICT
for obj_name in grand_dict:
    neighbors = []
    for other_name in grand_dict:
        if other_name != obj_name: #sounds good up until here
            if check_overlap(grand_dict[other_name][3], grand_dict[obj_name][3]):
                neighbors.append(other_name)
    grand_dict[obj_name].append(neighbors)

###################################################################################################
###################################################################################################

# STEP 1 DELIVERABLES GENERATOR
data_list = []
for key in grand_dict:
    data_dict = {
        "intensity": grand_dict[key][0],
        "name": key,
        "(x,y) of CoM": grand_dict[key][1],
        "pixel area": grand_dict[key][2],
        "L/R corners": grand_dict[key][3],
        "MBR Diag": grand_dict[key][4],
        "Intersectors": grand_dict[key][6]
    }
    data_list.append(data_dict)

df = pd.DataFrame(data_list)
html = df.to_html(index=False, bold_rows=True)

with open("step1.html", "w") as f:
    f.write(html)
```

Step 2: Describing Shape

Below is a chart of the output I got when using the code you'll see right below it. First, you'll see the block in the main function that generates this information for every contour. Following that excerpt, you'll see an overview of each of the helper functions referenced in that main block.

name	size	aspect ratio	shape	confusion
Carman	Small	wide	rectangle	[Kent, Dodge, Buell, EarlHall, StPaulChapel, Pupin]
ButlerLibrary	Large	wide	rectangle	[]
Lerner	Medium	wide	rectangle	[CollegeWalk]
Hamilton&Hartley&Wallach&JohnJay	Largest	narrow	C-shaped	[]
Journalism&Furnald	Large	medium-width	L-shaped	[]
CollegeWalk	Medium	wide	rectangle	[Lerner]
Kent	Small	wide	rectangle	[Carman, Dodge, Buell, EarlHall, StPaulChapel, Pupin]
Dodge	Small	wide	rectangle	[Carman, Kent, Buell, EarlHall, StPaulChapel, Pupin]
AlmaMater	Smallest	medium-width	rectangle	[]
Buell	Small	wide	rectangle	[Carman, Kent, Dodge, EarlHall, StPaulChapel, Pupin]
Philosophy	Small	narrow	I-shaped	[Lewisohn, Mathematics]
Lewisohn	Small	narrow	I-shaped	[Philosophy, Mathematics]
EarlHall	Small	wide	rectangle	[Carman, Kent, Dodge, Buell, StPaulChapel, Pupin]
StPaulChapel	Small	wide	rectangle	[Carman, Kent, Dodge, Buell, EarlHall, Pupin]
LowLibrary	Medium	medium-width	rectangle	[]
Mathematics	Small	narrow	I-shaped	[Philosophy, Lewisohn]
Fayerweather	Small	narrow	rectangle	[Avery, OldComputerCenter, NorthwestCorner]
Avery	Small	narrow	rectangle	[Fayerweather, OldComputerCenter, NorthwestCorner]
OldComputerCenter	Small	narrow	rectangle	[Fayerweather, Avery, NorthwestCorner]
Chandler&Havemeyer	Large	medium-width	rectangle	[]
Schermerhorn	Large	medium-width	C-shaped	[]
Uris	Large	narrow	rectangle	[]
NorthwestCorner	Small	narrow	rectangle	[Fayerweather, Avery, OldComputerCenter]
Mudd&EngTerrace&Fairchild&CS	Large	medium-width	asymmetrical	[]
SchapiroCEPSR	Small	medium-width	rectangle	[]
Pupin	Small	wide	rectangle	[Carman, Kent, Dodge, Buell, EarlHall, StPaulChapel]

```
# ADD SHAPES TO DICTIONARY
for i, contour in enumerate(contours):

    '''
    code seen above in the overview of Step 1
    '''

    #CLASSIFY BY SIZE, ASPECT RATIO, AND SHAPE
    what = []
    size = get_box_size(grand_dict[obj_name][2])
    what.append(size)
    aspect = aspect_ratio(grand_dict[obj_name][3])
    what.append(aspect)
    shape = shape_category(contour, image)
    what.append(shape)
    grand_dict[obj_name].append(what)
```

Size

This algorithm compares MBR areas to determine which size a building is. As you can see, the cutoffs for each "bucket" of sizing are as shown below in the code.

The reason why I opted to use box area rather than box diagonal is because the area of a rectangle is directly proportional to its size, and it represents the amount of space that the rectangle covers. Comparing the areas of two rectangles would allow you to determine which one occupies more space, and thus which one is "bigger".

The diagonal of a rectangle is also related to its size, but it may not be as accurate as the area for comparing the sizes of rectangles. The diagonal is affected not only by the size of the rectangle, but also by its aspect ratio (the ratio of its length to its width). Therefore, two rectangles with the same area but different aspect ratios may have different diagonals, which could lead to incorrect comparisons of their sizes.

```
#HELPER FUNCTION FOR BOX_SIZE
def get_box_size(area):
    if area <= 2000:
        if area <=200:
            return "Smallest"
        else:
            return "Small"
    elif area > 2000 and area <= 5000:
        return "Medium"
    elif area > 5000:
        if area > 12000:
            return "Largest"
        else:
            return "Large"
```

Aspect Ratio

This function takes a box as input, where a box is defined by two points that represent opposite corners of a contour's MBR. The function first calculates the width and height of the rectangle using the coordinates of the two points, and then calculates the diagonal length of the rectangle using the Pythagorean theorem. The aspect ratio of the rectangle is calculated by dividing the width by the height. The function returns one of three possible strings based on the aspect ratio of the rectangle: "narrow" if the aspect ratio is less than 0.75, "wide" if the aspect ratio is greater than 1.33, or "medium-width", indicating a rectangle of medium proportions between those two.

```
#HELPER FUCNTION FOR BOX ASPECT RATIO
def aspect_ratio(box):
    width = abs(box[0][0] - box[1][0])
    height = abs(box[0][1] - box[1][1])
    diagonal = int(np.sqrt((width ** 2) + (height ** 2)))

    if height != 0:
        aspect_ratio = width / height
    else:
        aspect_ratio = 0

    if aspect_ratio < 0.75:
        return "narrow"
    elif aspect_ratio > 1.33:
        return "wide"
    else:
        return "medium-width"
```

Shape

Below, you'll see two functions. The first one is implemented within the second. The first function takes a matrix (in the form of a NumPy array) as input and returns a boolean value indicating whether the matrix contains more than half black pixels. This value is needed for the second function to operate.

The **shape_category()** function takes a contour and an image as input, and returns a string representing the category of shape that the contour belongs to. First, the function calculates the bounding rectangle around the contour using the **cv2.boundingRect()** function. It then extracts the part of the input image that corresponds to the bounding rectangle around the contour, and calculates the width and height of each grid cell. The function then creates 12 sub-arrays (sec1 through sec12) to hold the pixels of each section of the grid. Each sub-array corresponds to one of the 12 cells in the grid, with the cell indices ranging from 1 to 12 as follows:

1	2	3	4
5	6	7	8
9	10	11	12

The function then checks the properties of each section of the grid to determine the category of shape that the contour belongs to. This is done using the **bp()** function, which checks whether a matrix contains more than half black pixels. Because the shapes are not perfect and the occasional corner or facade bleeds into a tile, I opted to search for "more than half" black pixels rather than "contains only black pixels" to ease the threshold used to consider a tile "emtpy".

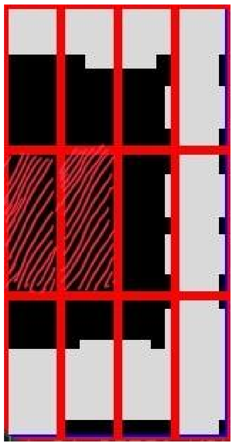


FIG 1: If sec5 and sec6 of the grid are dark, the shape is classified as "C-shaped"

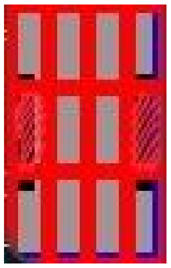


FIG 2: If sec5 and sec8 of the grid are dark, the shape is classified as "I-shaped". More is explained on this in the section below these figures.

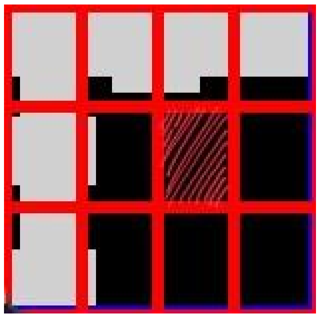


FIG 3: If sec7 of the grid is dark, the shape is classified as "L-shaped".

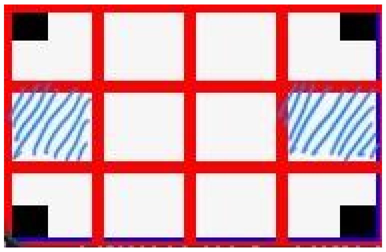


FIG 4: If sec5 and sec8 of the grid are <u>not</u> dark, the shape is classified as a "rectangle".

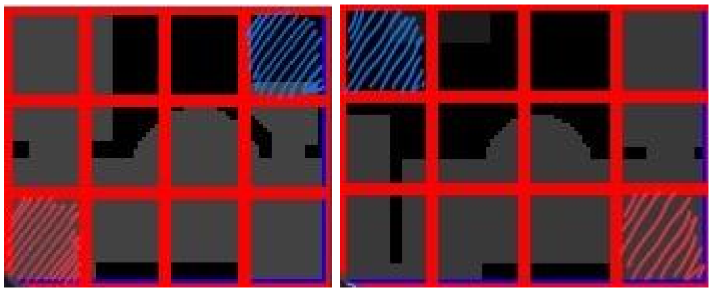


FIG 5: If sec1 and sec12 or sec4 and sec9 of the grid have <u>different</u> darkness levels, the shape is classified as "asymmetrical".

The reason why I decided to divide each MBR according to these specifications is because of the standard composition of the buildings on Columbia's campus. As seen in Fig 2, I chose this distribution of sections because the top and bottom sections of the "I" shape of all three of Columbia's I-shaped buildings fall cleanly into the top and bottom thirds of their MBR. Additionally, the main central column of the "I" shape fell into the middle two columns of its MBR when divided into fourths vertically. I based the MBR division around this particular shape given the harmony between these columns and this shape, then created section conditions for the other shapes based on this framework.

```
#HELPER FUNCTION FOR SHAPE CONFIGURATION - MAJORITY BLACK PIXELS
def bp(matrix):
    black_pixels = np.sum(matrix == 0)
    total_pixels = matrix.size
    return black_pixels > (total_pixels / 2)
```

```
#HELPER FUNCTION FOR CONTOUR SHAPE
def shape_category(contour, image):

    x, y, w, h = cv2.boundingRect(contour) # Find the bounding rectangle around the contour
    rect = image[y:y+h, x:x+w] # Extract the part of the input image corrensponding to the
rectangle
    cell_width = w // 4 # Calculate the width and height of each grid cell
    cell_height = h // 3

    # Create arrays to hold the pixels of each section
    sec1 = rect[0:cell_height, 0:cell_width]
    sec2 = rect[0:cell_height, cell_width:cell_width*2]
    sec3 = rect[0:cell_height, cell_width*2:cell_width*3]
    sec4 = rect[0:cell_height, cell_width*3:w]
    sec5 = rect[cell_height:cell_height*2, 0:cell_width]
    sec6 = rect[cell_height:cell_height*2, cell_width:cell_width*2]
    sec7 = rect[cell_height:cell_height*2, cell_width*2:cell_width*3]
    sec8 = rect[cell_height:cell_height*2, cell_width*3:w]
    sec9 = rect[cell_height*2:cell_height*3, 0:cell_width]
    sec10 = rect[cell_height*2:cell_height*3, cell_width:cell_width*2]
    sec11 = rect[cell_height*2:cell_height*3, cell_width*2:cell_width*3]
    sec12 = rect[cell_height*2:cell_height*3, cell_width*3:w]

    #determine the specs for each shape (more explanation in writeup)
    if (bp(sec5) and bp(sec6)):
        return "C-shaped"
    elif (bp(sec5) and bp(sec8)):
        return "I-shaped"
    elif (bp(sec7)):
        return "L-shaped"
    elif (bp(sec5) == False and bp(sec8) == False):
        return "rectangle"
    elif ( bp(sec1)!= bp(sec12) or bp(sec9)!= bp(sec4)):
        return "asymmetrical"
```

Note to Grader
Mentioned in the Project Instructions is the idea of using booleans to detect shape: *You should design, document, code, and test this by having a method or function for each shape description that evaluates a shape, and returns a boolean value for the presence of that shape.* I thought to have the shape detection function return a string rather than a boolean so I could use one function to test for all shapes in one step, rather than test each contour for every shape that it could be by using a boolean for each shape individually. I took this alternative route in the name of brevity.

Confusion

This loop iterates through grand_dict. For each item in the dictionary, the loop initializes an empty list called "twins". Then it proceeds to iterate through the dictionary again with a nested loop, this time examining all other items except for the current one. For each other item being examined, the loop checks whether it has the same value as the current item for a specific element (in this case, the 6th element of the value associated with each key which is an array holding each one of the geometric descriptors we've computed thus far). If the two arrays are identical, the name of the other item is appended to the twins list. Once all other items have been examined, the twins list is added as a new element to the value associated with the current item in grand_dict. The end result is a dictionary that contains all the same elements as the original grand_dict, but with an additional element for each item that lists the names of all the "twins" in the dictionary, or the other building names with identical geometric descritor arrays. This fills the last column on the output chart seen earlier.

```
#CREATE SHAPE-TWINS FOR EACH KEY IN GRAND_DICT
for obj_name in grand_dict:
    twins = []
    for other_name in grand_dict:
        if other_name != obj_name:
            if (grand_dict[obj_name][5] == grand_dict[other_name][5]):
                twins.append(other_name)
    grand_dict[obj_name].append(twins)
```

Minimization

As stated in *Ed Discussion Post #192*, "In this instance, the descriptions cannot be minimized". Therefore, I have not attempted to minimize the descriptors at this point with the exception of "smallest" and "largest".

Step 3: Describing Absolute Space

Below the output chart is the code that created it. Below that are the helper functions implemented.

name	verticality	horizontality	orientation	confusion
Carman	lowermost	leftmost	horizontally-oriented	[Lerner]
ButlerLibrary	lowermost	mid-width	horizontally-oriented	[]
Lerner	lowermost	leftmost	horizontally-oriented	[Carman]
Hamilton&Hartley&Wallach&JohnJay	lowermost	rightmost	vertically-oriented	[]
Journalism&Furnald	lower	leftmost	non-oriented	[]
CollegeWalk	lower	mid-width	horizontally-oriented	[]
Kent	lower	rightmost	horizontally-oriented	[]
Dodge	lower	leftmost	horizontally-oriented	[]
AlmaMater	mid-height	mid-width	non-oriented	[LowLibrary]
Buell	mid-height	right	horizontally-oriented	[]
Philosophy	mid-height	rightmost	vertically-oriented	[]
Lewisohn	mid-height	leftmost	vertically-oriented	[]
EarlHall	mid-height	leftmost	horizontally-oriented	[]
StPaulChapel	mid-height	rightmost	horizontally-oriented	[]
LowLibrary	mid-height	mid-width	non-oriented	[AlmaMater]
Mathematics	upper	leftmost	vertically-oriented	[]
Fayerweather	upper	rightmost	vertically-oriented	[]
Avery	upper	right	vertically-oriented	[]
OldComputerCenter	upper	left	vertically-oriented	[]
Chandler&Havemeyer	upper	leftmost	horizontally-oriented	[]
Schermerhorn	upper	rightmost	horizontally-oriented	[]
Uris	uppermost	mid-width	vertically-oriented	[]
NorthwestCorner	uppermost	leftmost	vertically-oriented	[]
Mudd&EngTerrace&Fairchild&CS	uppermost	right	horizontally-oriented	[]
SchapiroCEPSR	uppermost	mid-width	horizontally-oriented	[]
Pupin	uppermost	left	horizontally-oriented	[]

```
# ADD SHAPES TO DICTIONARY
for i, contour in enumerate(contours):

    '''
    code seen above in the overview of Steps 1 and 2
    '''

    #CLASSIFY BY LOCATION AND ORIENTATION
    where = []
    vert = get_vert_section(image, contour)
    where.append(vert)
    hoz = get_hoz_section(image, contour)
    where.append(hoz)
    ornt = get_orientation(contour)
    where.append(ornt)
    grand_dict[obj_name].append(where)
```

Verticality

I apologize for the small font size of the code. This is due to the length of the if-statement conditions. The function takes in two parameters: the image and the contour. The image height and width are determined using the shape method of the image object. The height is divided into five equal horizontal sections, and each section is assigned a name that corresponds to its position in the image. The bounding rectangle of the contour is then calculated using the OpenCV function **cv2.boundingRect()**, along with calculating its center. The function checks which section the center of the bounding rectangle falls within by comparing its x and y coordinates to the bounds of each section. If the center falls within a section, the function returns the name of that section.

```
#HELPER FUNCTION FOR VERTICAL LOCATION
def get_vert_section(image, contour):

    # Get the height and width of the input image
    height, width = image.shape[:2]

    # Calculate the bounds of each section
    uppermost_bounds = (0, 0, width, height // 5)
    upper_bounds = (0, height // 5, width, height // 5)
    mid_height_bounds = (0, 2 * height // 5, width, height // 5)
    lower_bounds = (0, 3 * height // 5, width, height // 5)
    lowermost_bounds = (0, 4 * height // 5, width, height // 5)

    # Get the bounding rectangle of the contour
    x, y, w, h = cv2.boundingRect(contour)

    # Calculate the center of the bounding rectangle
    center_x = x + w // 2
    center_y = y + h // 2

    # Check which section the center of the bounding rectangle falls within
    if center_y >= uppermost_bounds[1] and center_y <= uppermost_bounds[1] + uppermost_bounds[3] and center_x >= uppermost_bounds[0] and center_x <= uppermost_bounds[0] + uppermost_bounds[2]:
        return 'uppermost'
    elif center_y >= upper_bounds[1] and center_y <= upper_bounds[1] + upper_bounds[3] and center_x >= upper_bounds[0] and center_x <= upper_bounds[0] + upper_bounds[2]:
        return 'upper'
    elif center_y >= mid_height_bounds[1] and center_y <= mid_height_bounds[1] + mid_height_bounds[3] and center_x >= mid_height_bounds[0] and center_x <= mid_height_bounds[0] + mid_height_bounds[2]:
        return 'mid-height'
    elif center_y >= lower_bounds[1] and center_y <= lower_bounds[1] + lower_bounds[3] and center_x >= lower_bounds[0] and center_x <= lower_bounds[0] + lower_bounds[2]:
        return 'lower'
    elif center_y >= lowermost_bounds[1] and center_y <= lowermost_bounds[1] + lowermost_bounds[3] and center_x >= lowermost_bounds[0] and center_x <= lowermost_bounds[0] + lowermost_bounds[2]:
        return 'lowermost'
    else:
        return 'unknown'
```

Horizontality

This function has exactly the same functionality with the exception of dividing the image vertically rather than horizontally.

```
#HELPER FUNCTION FOR HORIZONTAL LOCATION
def get_hoz_section(image, contour):
    # Get the height and width of the input image
    height, width = image.shape[:2]

    # Calculate the bounds of each section
    leftmost_bounds = (0, 0, width // 5, height)
    left_bounds = (width // 5, 0, width // 5, height)
    mid_width_bounds = (2 * width // 5, 0, width // 5, height)
    right_bounds = (3 * width // 5, 0, width // 5, height)
    rightmost_bounds = (4 * width // 5, 0, width // 5, height)

    # Get the bounding rectangle of the contour
    x, y, w, h = cv2.boundingRect(contour)

    # Calculate the center of the bounding rectangle
    center_x = x + w // 2
    center_y = y + h // 2

    # Check which section the center of the bounding rectangle falls within
    if center_x >= leftmost_bounds[0] and center_x <= leftmost_bounds[0] + leftmost_bounds[2] and center_y >= leftmost_bounds[1] and center_y <= leftmost_bounds[1] + leftmost_bounds[3]:
        return 'leftmost'
    elif center_x >= left_bounds[0] and center_x <= left_bounds[0] + left_bounds[2] and center_y >= left_bounds[1] and center_y <= left_bounds[1] + left_bounds[3]:
        return 'left'
    elif center_x >= mid_width_bounds[0] and center_x <= mid_width_bounds[0] + mid_width_bounds[2] and center_y >= mid_width_bounds[1] and center_y <= mid_width_bounds[1] + mid_width_bounds[3]:
        return 'mid-width'
    elif center_x >= right_bounds[0] and center_x <= right_bounds[0] + right_bounds[2] and center_y >= right_bounds[1] and center_y <= right_bounds[1] + right_bounds[3]:
        return 'right'
    elif center_x >= rightmost_bounds[0] and center_x <= rightmost_bounds[0] + rightmost_bounds[2] and center_y >= rightmost_bounds[1] and center_y <= rightmost_bounds[1] + rightmost_bounds[3]:
        return 'rightmost'
    else:
        return 'unknown'
```

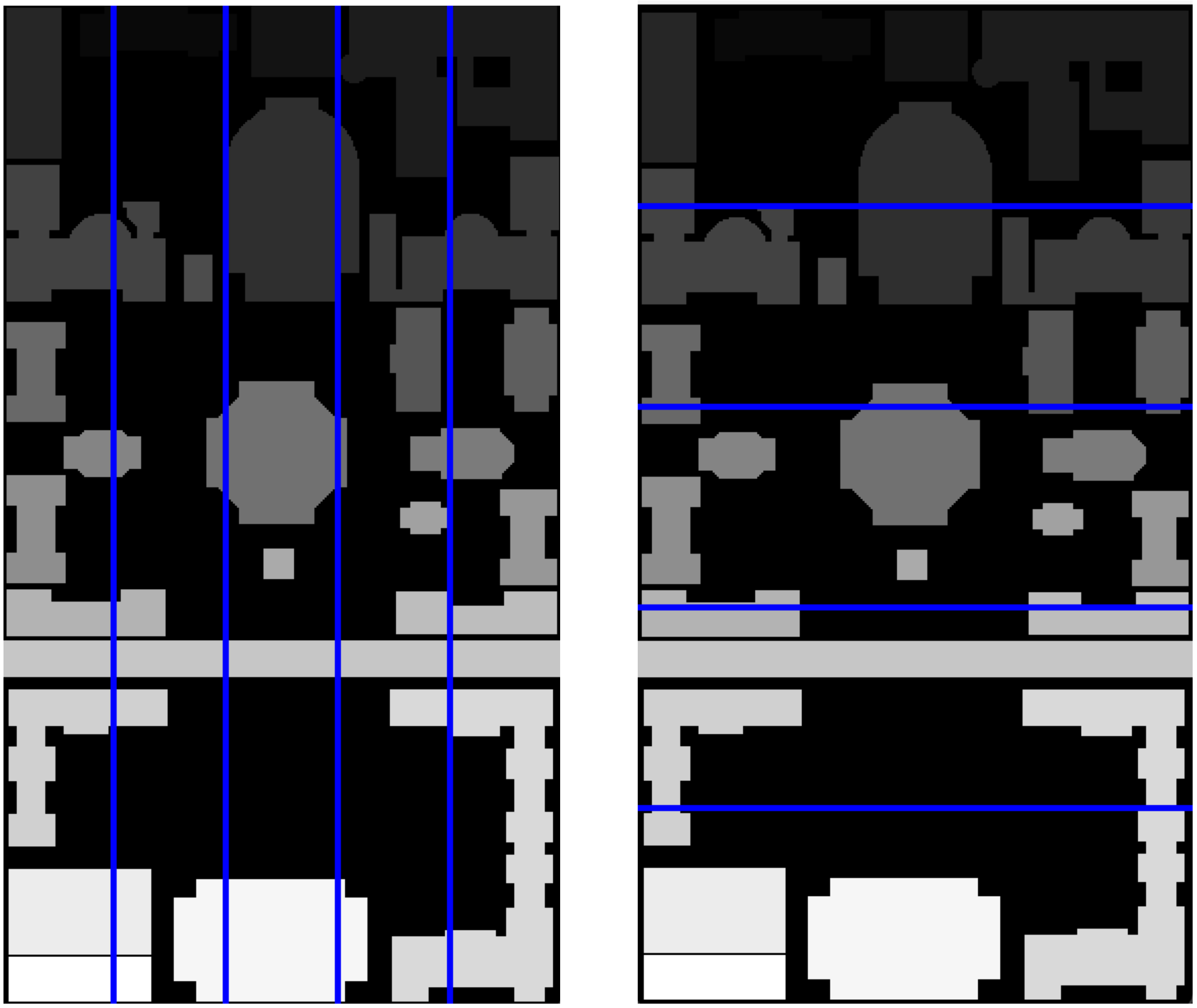


FIG 6: On the left are the divisions used for get_hoz_section(), and on the right are the divisions used for get_vert_section().

Orientation

The function calculates the bounding rectangle of the contour using **cv2.boundingRect()**, then calculates the aspect ratio of the bounding rectangle by dividing its width by its height. It checks if the aspect ratio is within a threshold of 1.0, which represents a non-oriented object. If the aspect ratio is close enough* to 1.0, the function returns 'non-oriented'. If the aspect ratio is greater than 1.0, the function determines that the object is horizontally-oriented, which means it is wider than it is tall. If the aspect ratio is less than 1.0, the function determines that the object is vertically-oriented, which means it is taller than it is wide.

*The threshold parameter, currently set to 0.1, can be adjusted to increase or decrease the tolerance for what is considered a non-oriented object. This means that an aspect ratio between 0.9 and 1.1 will be cateogorized as non-oriented.

```
#HELPER FUNCTION FOR ORIENTATION
def get_orientation(contour, threshold=0.1):

    # Get the bounding rectangle of the contour
    x, y, w, h = cv2.boundingRect(contour)
    # Calculate the aspect ratio of the bounding rectangle
    aspect_ratio = w / h

    # Check if the aspect ratio is within the threshold of 1.0 (non-oriented)
    if abs(aspect_ratio - 1.0) < threshold:
        return 'non-oriented'
    # Check if the aspect ratio is greater than 1.0 (horizontally-oriented)
    elif aspect_ratio > 1.0:
        return 'horizontally-oriented'
    # Otherwise, the aspect ratio is less than 1.0 (vertically-oriented)
    else:
        return 'vertically-oriented'
```

Confusion

At this point in the code, I decided to create a function to standardize the “confusion” array generation process. This algorithm works exactly the same as the one mentioned under “Confusion” for Step 2:

```
#FUNCTION TO CREATE "CONFUSION" ARRAYS FOR A GIVEN METRIC
def get_twins(metric):
    for obj_name in grand_dict:
        twins = []
        for other_name in grand_dict:
            if other_name != obj_name:
                if (grand_dict[obj_name][metric] == grand_dict[other_name][metric]):
                    twins.append(other_name)
        grand_dict[obj_name].append(twins)
```

The accompanying line for this metric was as such, as the “where” array was housed in index position 6 in each key’s value array:

```
get_twins(6) # should add item arr[9] to the value array being the list of location twins
```

Step 4: Describing Relative Space

The presentation of the code used to contrive this output will be in the same order the previous explanations were in.

name	near to
Carman	[ButlerLibrary, Lerner]
ButlerLibrary	[Carman, Lerner, Hamilton&Hartley&Wallach&JohnJay, Journalism&Furnald]
Lerner	[Carman, ButlerLibrary, Journalism&Furnald]
Hamilton&Hartley&Wallach&JohnJay	[ButlerLibrary, CollegeWalk, Kent]
Journalism&Furnald	[ButlerLibrary, Lerner, CollegeWalk, Dodge]
CollegeWalk	[Hamilton&Hartley&Wallach&JohnJay, Journalism&Furnald, Kent, Dodge, Philosophy, Lewisohn]
Kent	[Hamilton&Hartley&Wallach&JohnJay, CollegeWalk, Philosophy]
Dodge	[Journalism&Furnald, CollegeWalk, Lewisohn]
AlmaMater	[LowLibrary]
Buell	[StPaulChapel]
Philosophy	[CollegeWalk, Kent, StPaulChapel]
Lewisohn	[CollegeWalk, Dodge, EarlHall]
EarlHall	[Lewisohn, Mathematics]
StPaulChapel	[Buell, Philosophy, Fayerweather, Avery]
LowLibrary	[Avery]
Mathematics	[EarlHall, Chandler&Havemeyer]
Fayerweather	[StPaulChapel, Schermerhorn]
Avery	[StPaulChapel, LowLibrary, Schermerhorn, Uris]
OldComputerCenter	[Chandler&Havemeyer, Uris]
Chandler&Havemeyer	[Mathematics, OldComputerCenter, Uris, NorthwestCorner]
Schermerhorn	[Fayerweather, Avery, Uris, Mudd&EngTerrace&Fairchild&CS]
Uris	[Avery, OldComputerCenter, Chandler&Havemeyer, Schermerhorn, Mudd&EngTerrace&Fairchild&CS, SchapiroCEPSR, Pupin]
NorthwestCorner	[Chandler&Havemeyer, Pupin]
Mudd&EngTerrace&Fairchild&CS	[Schermerhorn, Uris, SchapiroCEPSR]
SchapiroCEPSR	[Uris, Mudd&EngTerrace&Fairchild&CS, Pupin]
Pupin	[Uris, NorthwestCorner, SchapiroCEPSR]

The line of code shown below was added to the initial raw data generation code for each contour. It saves the contour data for every contour as an item in each key's value array:

```
# ADD SHAPES TO DICTIONARY
for i, contour in enumerate(contours):

    '''
    code seen above in the overview of Steps 1, 2, and 3
    '''

    grand_dict[obj_name].append(contour)
```

This block below is looping through each key in grand_dict and creating a list of nearness values for each key. It does this by comparing the contour of the current key (obj_name) to the contours of all other keys in the dictionary (other_name) using the **calculate_nearness()** function.

If the nearness value is below a certain threshold (0.165), the other_name key is added to the nearness list for the current obj_name key. If the obj_name key is "AlmaMater", the nearness threshold is a bit higher (0.3) before adding the other_name key to the nearness list.

Finally, the nearness list for each key is appended to the value of that key in the grand_dict dictionary (at index 11) so that it can be accessed later

```python
#  CREATE NEARNESS ARRAY FOR EACH KEY IN GRAND_DICT
for obj_name in grand_dict:
    nearness = []
    for other_name in grand_dict:
        if other_name != obj_name:
            near = calculate_nearness(grand_dict[other_name][7], grand_dict[obj_name][7])
            if obj_name == "AlmaMater":
                if near < 0.3:
                    nearness.append(other_name)
            if near < 0.165:
                nearness.append(other_name)
    grand_dict[obj_name].append(nearness)
```

The threshold for “AlmaMater” was isolated and specified to be higher than the threshold used for the rest because, according to the computation in **calculate_nearness()**, this was the minimum setting to allow “LowLibrary” to appear in its nearness array.

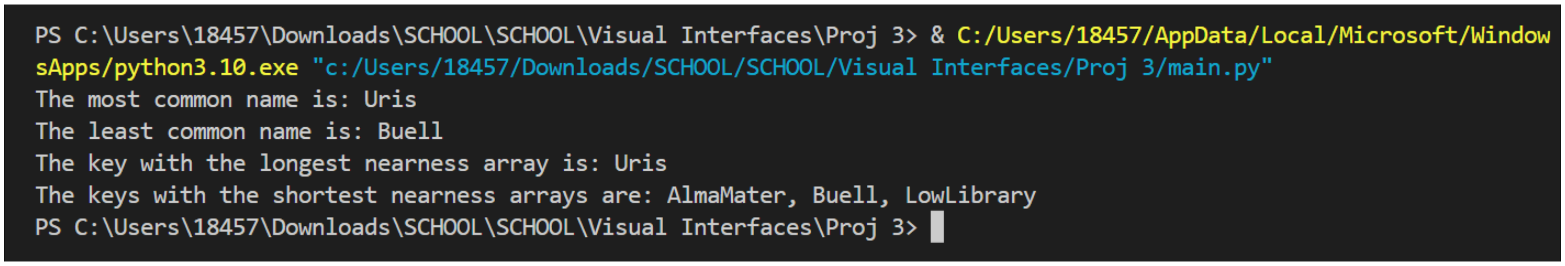
Nearness

This function takes two contours as input arguments. It first calculates the bounding rectangles for the two contours using **cv2.boundingRect()** function. Then, it calculates the points along the two bounding rectangles by getting the four corners of the rectangle. After that, the function loops through every pair of points from the two sets of points and calculates the Euclidean distance between them using **np.linalg.norm()**. It keeps track of the minimum distance found so far and finally, normalizes the minimum distance between 0 and 1 by dividing it by the maximum possible distance between two points in the rectangles. Finally, the function returns the normalized distance. I chose to normalize the distance to a common range between 0 and 1 to make it easier to compare the distances.

```python
def calculate_nearness(contour1, contour2):

    # Create bounding rectangles for the contours
    x1, y1, w1, h1 = cv2.boundingRect(contour1)
    x2, y2, w2, h2 = cv2.boundingRect(contour2)

    # Calculate the points along the bounding rectangles
    rect1_pts = [(x1, y1), (x1+w1, y1), (x1, y1+h1), (x1+w1, y1+h1)]
    rect2_pts = [(x2, y2), (x2+w2, y2), (x2, y2+h2), (x2+w2, y2+h2)]

    # Calculate the minimum distance between the two bounding rectangles
    min_dist = float('inf')
    for pt1 in rect1_pts:
        for pt2 in rect2_pts:
            dist = np.linalg.norm(np.array(pt1) - np.array(pt2))
            if dist < min_dist:
                min_dist = dist

    # Normalize the distance between 0 and 1 based on the maximum possible distance
    max_dist = np.linalg.norm(np.array([w1+h1, w2+h2]))
    norm_dist = min_dist / max_dist

    return norm_dist
```

Confusion

To answer the question of *Find which source S is nearTo(S, T) for the most targets T, then find the S for the least targets T. Conversely, find which target T is nearTo(S, T) for the most sources S, then find the T for the least.sources S*, I've prepared such metrics using simple algorithms used to return data on the content of the nearness arrays for each key, which can be seen above in the output chart I provided.



```
PS C:\Users\18457\Downloads\SCHOOL\SCHOOL\Visual Interfaces\Proj 3> & C:/Users/18457/AppData/Local/Microsoft/Window
sApps/python3.10.exe "c:/Users/18457/Downloads/SCHOOL/SCHOOL/Visual Interfaces/Proj 3/main.py"
The most common name is: Uris
The least common name is: Buell
The key with the longest nearness array is: Uris
The keys with the shortest nearness arrays are: AlmaMater, Buell, LowLibrary
PS C:\Users\18457\Downloads\SCHOOL\SCHOOL\Visual Interfaces\Proj 3> 
```

Step 5: Total Descriptions

I used the following algorithm to detect if at least one of the following three arrays were full so I could include at least one metric of comparison between two buildings for each building. This returned "yes" for all buildings, meaning each building description would be able to include at least one "near to" or "next to" metric.

```
def check_full(arr1, arr2, arr3):
    if arr1 or arr2 or arr3:
        return "Yes"
    else:
        return "No"

for obj_name in grand_dict:
    arr1 = grand_dict[obj_name][8]  #neighbors
    arr2 = grand_dict[obj_name][10] #location twins
    arr3 = grand_dict[obj_name][11] #nearArray
    print(check_full(arr1, arr2, arr3))
```

This was the code I used to generate the output you'll see in the chart below it. This function takes in six arguments: a string building and five arrays arr1, arr2, arr3, arr4, and arr5. It first creates an empty array called nearTo. It then checks if arr3 has any values, and if it does, it copies the first two values into the nearTo array. If arr3 is empty or only has one value, the function checks arr4 for values to add to nearTo, again copying one or two values depending on whether arr3 already had one value or not. If arr3 and arr4 are both empty or only have one value, the function checks arr5 for values to add to nearTo, again copying one or two values as necessary.

The function then creates a string called output, which contains the building argument, arr1, and arr2. It appends to this string the phrase "building near to" and then the first value in nearTo (if it has any values). If nearTo has more than one value, it also appends "and" and the second value in nearTo. Finally, the function returns the formatted output string.

```
arr1 = grand_dict[obj_name][5]   #shape descriptor
arr2 = grand_dict[obj_name][6]   #location descriptor
arr3 = grand_dict[obj_name][8]   #neighbors
arr4 = grand_dict[obj_name][10]  #location twins
arr5 = grand_dict[obj_name][11]  #nearArray
```

I chose to search the descritor arrays 3-5 in this priority order because overlapping MBRs is a failsafe way to consider something "near" to something else. After that, I prioritized the location confusion array as it provides a more absolute measure compared to the relative "nearness" computation because based on the algorithm and thresholds I used as seen in **Step 4**, the "location twins" metric is more reliable. For instance, based on the **get_twins(6)**, the line that returns information describing which buildings have identical location identifier arrays, Low Library is determined to be near Alma, while my nearness computation states that Low is near Avery, which is not as intuitive from a human perspective.

```
#FUNCTION TO FORMAT OUTPUT STRING

    # Create nearTo array
    nearTo = []
    if arr3:
        nearTo = arr3[:2]
    elif arr4:
        nearTo = arr4[:2]
    elif arr5:
        nearTo = arr5[:2]

    # Create output string
    output = f"{building}: {arr1[0]}, {arr1[1]}, {arr1[2]}, {arr2[0]}, {arr2[1]}, {arr2[2]} building near to "
    if nearTo:
        output += f"{nearTo[0]}"
    if len(nearTo) > 1:
        output += f" and {nearTo[1]}"
    return output

#USED TO CREATE OUTPUT HMTL FILE
with open("output.html", "w") as f:
    for obj_name in grand_dict:
        arr1 = grand_dict[obj_name][5]   #shape descriptor
        arr2 = grand_dict[obj_name][6]   #location descriptor
        arr3 = grand_dict[obj_name][8]   #neighbors
        arr4 = grand_dict[obj_name][10]  #location twins
        arr5 = grand_dict[obj_name][11]  #nearArray
        building = obj_name
        output = building_near(building, arr1, arr2, arr3, arr4, arr5)
        output = output.replace(building, f"<b>{building}</b>")
        f.write(f"{output}<br>")
```

Carman: Small, wide, rectangular, lowermost, leftmost, horizontally-oriented building near to Lerner
ButlerLibrary: Large, wide, rectangular, lowermost, mid-width, horizontally-oriented building near to Carman and Lerner
Lerner: Medium, wide, rectangular, lowermost, leftmost, horizontally-oriented building near to Carman
Hamilton&Hartley&Wallach&JohnJay: Largest, narrow, C-shaped, lowermost, rightmost, vertically-oriented building near to ButlerLibrary and CollegeWalk
Journalism&Furnald: Large, medium-width, L-shaped, lower, leftmost, non-oriented building near to ButlerLibrary and Lerner
CollegeWalk: Medium, wide, rectangular, lower, mid-width, horizontally-oriented building near to Hamilton&Hartley&Wallach&JohnJay and Journalism&Furnald
Kent: Small, wide, rectangular, lower, rightmost, horizontally-oriented building near to Hamilton&Hartley&Wallach&JohnJay and CollegeWalk
Dodge: Small, wide, rectangular, lower, leftmost, horizontally-oriented building near to Journalism&Furnald and CollegeWalk
AlmaMater: Smallest, medium-width, rectangular, mid-height, mid-width, non-oriented building near to LowLibrary
Buell: Small, wide, rectangular, mid-height, right, horizontally-oriented building near to StPaulChapel
Philosophy: Small, narrow, I-shaped, mid-height, rightmost, vertically-oriented building near to CollegeWalk and Kent
Lewisohn: Small, narrow, I-shaped, mid-height, leftmost, vertically-oriented building near to EarlHall
EarlHall: Small, wide, rectangular, mid-height, leftmost, horizontally-oriented building near to Lewisohn
StPaulChapel: Small, wide, rectangular, mid-height, rightmost, horizontally-oriented building near to Buell and Philosophy
LowLibrary: Medium, medium-width, rectangular, mid-height, mid-width, non-oriented building near to AlmaMater
Mathematics: Small, narrow, I-shaped, upper, leftmost, vertically-oriented building near to EarlHall and Chandler&Havemeyer
Fayerweather: Small, narrow, rectangular, upper, rightmost, vertically-oriented building near to StPaulChapel and Schermerhorn
Avery: Small, narrow, rectangular, upper, right, vertically-oriented building near to StPaulChapel and LowLibrary
OldComputerCenter: Small, narrow, rectangular, upper, left, vertically-oriented building near to Chandler&Havemeyer and Uris
Chandler&Havemeyer: Large, medium-width, rectangular, upper, leftmost, horizontally-oriented building near to Mathematics and OldComputerCenter
Schermerhorn: Large, medium-width, rectangular, upper, rightmost, horizontally-oriented building near to Mudd&EngTerrace&Fairchild&CS
Uris: Large, narrow, rectangular, uppermost, mid-width, vertically-oriented building near to Mudd&EngTerrace&Fairchild&CS
NorthwestCorner: Small, narrow, rectangular, uppermost, leftmost, vertically-oriented building near to Chandler&Havemeyer and Pupin
Mudd&EngTerrace&Fairchild&CS: Large, medium-width, asymmetrical, uppermost, right, horizontally-oriented building near to Schermerhorn and Uris
SchapiroCEPSR: Small, medium-width, rectangular, uppermost, mid-width, horizontally-oriented building near to Uris and Mudd&EngTerrace&Fairchild&CS
Pupin: Small, wide, rectangular, uppermost, left, horizontally-oriented building near to Uris and NorthwestCorner

The reason why I included the names of buildings in the “near to” portion is that, when perusing this reference as a whole, and if you’re standing on campus, the proximity of one building to another will help you get a hold on what buildings are named what. Paired with the shape and location descriptors, if you’re seeing the “smallest, medium-width, rectangular, mid-height, mid-width, non-oriented building near to LowLibrary”, then go to the description for Low Library, unaware of where it is, “medium, medium-width, rectangular, mid-height, mid-width, non-oriented building near to AlmaMater”, you’re able to deduce the names of two buildings at once given they’re relationship to each other.

This was my reasoning for inserting building names into the nearness portion of the output. It does a faster job at informing the reader of campus building names as well as keeps the building descriptions much shorter than if you had to include full-length descriptions for every building a certain building was near to.

Code Used

Below is a listing of all the code used for this assignment. You'll see all the algorithms I've explained above listed in the order I've implemented them inside the code. Some blocks may have evolved slightly between the time I first included them in this writeup to this final submission below, but use generally the same computations and thresholds.

```python
import cv2
import numpy as np
import pandas as pd
import math
from collections import Counter


#HELPER FUNCTION FOR SHAPE CONFIGURATION - MAJORITY BLACK PIXELS
def bp(matrix):
    black_pixels = np.sum(matrix == 0)
    total_pixels = matrix.size
    return black_pixels > (total_pixels / 2)

#HELPER FUNCTION FOR CONTOUR SHAPE
def shape_category(contour, image):

    x, y, w, h = cv2.boundingRect(contour) # Find the bounding rectangle around the contour
    rect = image[y:y+h, x:x+w] # Extract the part of the input image correnponding to the rectangle
    cell_width = w // 4 # Calculate the width and height of each grid cell
    cell_height = h // 3

    # Create arrays to hold the pixels of each section
    sec1 = rect[0:cell_height, 0:cell_width]
    sec2 = rect[0:cell_height, cell_width:cell_width*2]
    sec3 = rect[0:cell_height, cell_width*2:cell_width*3]
    sec4 = rect[0:cell_height, cell_width*3:w]
    sec5 = rect[cell_height:cell_height*2, 0:cell_width]
    sec6 = rect[cell_height:cell_height*2, cell_width:cell_width*2]
    sec7 = rect[cell_height:cell_height*2, cell_width*2:cell_width*3]
    sec8 = rect[cell_height:cell_height*2, cell_width*3:w]
    sec9 = rect[cell_height*2:cell_height*3, 0:cell_width]
    sec10 = rect[cell_height*2:cell_height*3, cell_width:cell_width*2]
    sec11 = rect[cell_height*2:cell_height*3, cell_width*2:cell_width*3]
    sec12 = rect[cell_height*2:cell_height*3, cell_width*3:w]

    #determine the specs for each shape (more explanation in writeup)
    if (bp(sec5) and bp(sec6)):
        return "C-shaped"
    elif (bp(sec5) and bp(sec8)):
        return "I-shaped"
    elif (bp(sec7)):
        return "L-shaped"
    elif (bp(sec5) == False and bp(sec8) == False):
        return "rectangular"
    elif ( bp(sec1)!= bp(sec12) or bp(sec9)!= bp(sec4)):
        return "asymmetrical"

#HELPER FUNCTION FOR BOX_SIZE
def get_box_size(area):
    if area <= 2000:
        if area <=200:
            return "Smallest"
        else:
            return "Small"
    elif area > 2000 and area <= 5000:
        return "Medium"
    elif area > 5000:
        if area > 12000:
            return "Largest"
        else:
            return "Large"

#HELPER FUCNTION FOR BOX ASPECT RATIO
def aspect_ratio(box):
    width = abs(box[0][0] - box[1][0])
    height = abs(box[0][1] - box[1][1])
    diagonal = int(np.sqrt((width ** 2) + (height ** 2)))

    if height != 0:
        aspect_ratio = width / height
    else:
        aspect_ratio = 0

    if aspect_ratio < 0.75:
        return "narrow"
    elif aspect_ratio > 1.33:
        return "wide"
    else:
        return "medium-width"

#HELPER FUNCTION FOR BOX OVERLAP
def check_overlap(rect1, rect2):

    rect1_ur = rect1[0]
    rect1_ll = rect1[1]
    rect2_ur = rect2[0]
    rect2_ll = rect2[1]

    # Check for overlap
    if rect1_ur[0] < rect2_ll[0] or rect2_ur[0] < rect1_ll[0]:
        return False # Rectangles don't overlap horizontally
    if rect1_ur[1] < rect2_ll[1] or rect2_ur[1] < rect1_ll[1]:
        return False # Rectangles don't overlap vertically
    return True

#HELPER FUNCTION FOR VERTICAL LOCATION
def get_vert_section(image, contour):

    # Get the height and width of the input image
    height, width = image.shape[:2]

    # Calculate the bounds of each section
    uppermost_bounds = (0, 0, width, height // 5)
    upper_bounds = (0, height // 5, width, height // 5)
    mid_height_bounds = (0, 2 * height // 5, width, height // 5)
    lower_bounds = (0, 3 * height // 5, width, height // 5)
    lowermost_bounds = (0, 4 * height // 5, width, height // 5)

    # Get the bounding rectangle of the contour
    x, y, w, h = cv2.boundingRect(contour)

    # Calculate the center of the bounding rectangle
    center_x = x + w // 2
    center_y = y + h // 2

    # Check which section the center of the bounding rectangle falls within
    if center_y >= uppermost_bounds[1] and center_y <= uppermost_bounds[1] + uppermost_bounds[3] and center_x >= uppermost_bounds[0] and center_x <= uppermost_bounds[0] + uppermost_bounds[2]:
        return 'uppermost'
    elif center_y >= upper_bounds[1] and center_y <= upper_bounds[1] + upper_bounds[3] and center_x >= upper_bounds[0] and center_x <= upper_bounds[0] + upper_bounds[2]:
        return 'upper'
    elif center_y >= mid_height_bounds[1] and center_y <= mid_height_bounds[1] + mid_height_bounds[3] and center_x >= mid_height_bounds[0] and center_x <= mid_height_bounds[0] + mid_height_bounds[2]:
        return 'mid-height'
    elif center_y >= lower_bounds[1] and center_y <= lower_bounds[1] + lower_bounds[3] and center_x >= lower_bounds[0] and center_x <= lower_bounds[0] + lower_bounds[2]:
        return 'lower'
    elif center_y >= lowermost_bounds[1] and center_y <= lowermost_bounds[1] + lowermost_bounds[3] and center_x >= lowermost_bounds[0] and center_x <= lowermost_bounds[0] + lowermost_bounds[2]:
        return 'lowermost'
    else:
        return 'unknown'

#HELPER FUNCTION FOR HORIZONTAL LOCATION
def get_hoz_section(image, contour):
    # Get the height and width of the input image
    height, width = image.shape[:2]

    # Calculate the bounds of each section
    leftmost_bounds = (0, 0, width // 5, height)
    left_bounds = (width // 5, 0, width // 5, height)
    mid_width_bounds = (2 * width // 5, 0, width // 5, height)
    right_bounds = (3 * width // 5, 0, width // 5, height)
    rightmost_bounds = (4 * width // 5, 0, width // 5, height)

    # Get the bounding rectangle of the contour
    x, y, w, h = cv2.boundingRect(contour)

    # Calculate the center of the bounding rectangle
    center_x = x + w // 2
    center_y = y + h // 2

    # Check which section the center of the bounding rectangle falls within
    if center_x >= leftmost_bounds[0] and center_x <= leftmost_bounds[0] + leftmost_bounds[2] and center_y >= leftmost_bounds[1] and center_y 
<= leftmost_bounds[1] + leftmost_bounds[3]:
        return 'leftmost'
    elif center_x >= left_bounds[0] and center_x <= left_bounds[0] + left_bounds[2] and center_y >= left_bounds[1] and center_y <= 
left_bounds[1] + left_bounds[3]:
        return 'left'
    elif center_x >= mid_width_bounds[0] and center_x <= mid_width_bounds[0] + mid_width_bounds[2] and center_y >= mid_width_bounds[1] and 
center_y <= mid_width_bounds[1] + mid_width_bounds[3]:
        return 'mid-width'
    elif center_x >= right_bounds[0] and center_x <= right_bounds[0] + right_bounds[2] and center_y >= right_bounds[1] and center_y <= 
right_bounds[1] + right_bounds[3]:
        return 'right'
    elif center_x >= rightmost_bounds[0] and center_x <= rightmost_bounds[0] + rightmost_bounds[2] and center_y >= rightmost_bounds[1] and 
center_y <= rightmost_bounds[1] + rightmost_bounds[3]:
        return 'rightmost'
    else:
        return 'unknown'

#HELPER FUNCTION FOR ORIENTATION
def get_orientation(contour, threshold=0.1):

    # Get the bounding rectangle of the contour
    x, y, w, h = cv2.boundingRect(contour)

    # Calculate the aspect ratio of the bounding rectangle
    aspect_ratio = w / h

    # Check if the aspect ratio is within the threshold of 1.0 (i.e., non-oriented)
    if abs(aspect_ratio - 1.0) < threshold:
        return 'non-oriented'
    # Check if the aspect ratio is greater than 1.0 (i.e., horizontally-oriented)
    elif aspect_ratio > 1.0:
        return 'horizontally-oriented'
    # Otherwise, the aspect ratio is less than 1.0 (i.e., vertically-oriented)
    else:
        return 'vertically-oriented'

#FUNCTION TO CREATE "CONFUSION" ARRAYS FOR A GIVEN METRIC
def get_twins(metric):
    for obj_name in grand_dict:
        twins = []
        for other_name in grand_dict:
            if other_name != obj_name:
                if (grand_dict[obj_name][metric] == grand_dict[other_name][metric]):
                    twins.append(other_name)
        grand_dict[obj_name].append(twins)

#FUNCTION TO COMPUTE NEARNESS FOR TWO CONTOURS
def calculate_nearness(contour1, contour2):

    # Create bounding rectangles for the contours
    x1, y1, w1, h1 = cv2.boundingRect(contour1)
    x2, y2, w2, h2 = cv2.boundingRect(contour2)

    # Calculate the points along the bounding rectangles
    rect1_pts = [(x1, y1), (x1+w1, y1), (x1, y1+h1), (x1+w1, y1+h1)]
    rect2_pts = [(x2, y2), (x2+w2, y2), (x2, y2+h2), (x2+w2, y2+h2)]

    # Calculate the minimum distance between the two bounding rectangles
    min_dist = float('inf')
    for pt1 in rect1_pts:
        for pt2 in rect2_pts:
            dist = np.linalg.norm(np.array(pt1) - np.array(pt2))
            if dist < min_dist:
                min_dist = dist

    # Normalize the distance between 0 and 1 based on the maximum possible distance
    max_dist = np.linalg.norm(np.array([w1+h1, w2+h2]))
    norm_dist = min_dist / max_dist

    return norm_dist

#FINAL OUTPUT FORMATTING FUNCTION
def building_near(building, arr1, arr2, arr3, arr4, arr5):
    # Create nearTo array
    nearTo = []
    if arr3:
        nearTo = arr3[:2]
    elif arr4:
        nearTo = arr4[:2]
    elif arr5:
        nearTo = arr5[:2]

    # Create output string
    output = f"{building}: {arr1[0]}, {arr1[1]}, {arr1[2]}, {arr2[0]}, {arr2[1]}, {arr2[2]} building near to "
    if nearTo:
        output += f"{nearTo[0]}"
    if len(nearTo) > 1:
        output += f" and {nearTo[1]}"
    return output

##################################################################################################
##################################################################################################

# INITIALIZE DICTIONARY
grand_dict = {}

# READ IN TABLE.TXT TO POPULATE NAMELIST
namelist = []
with open("Table.txt") as f:
    lines = f.readlines()
for line in lines:
    items = line.split() # Split the line by whitespace
    name = items[1] # Store the second item as a string labeled "name"
    namelist.append(name)
namelist = namelist[::-1] # Reverse namelist to match contour read-in order in the next block
i = 0

# READ IMAGE + LOCATE SHAPES
image = cv2.imread('Labeled.pgm', cv2.IMREAD_COLOR)
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
contours, _ = cv2.findContours(gray_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
image_height = image.shape[0]

# ADD SHAPES TO DICTIONARY
for i, contour in enumerate(contours):

    # FIND OBJECT
    x, y, w, h = cv2.boundingRect(contour)
    obj = image[y:y+h, x:x+w]

    # CALCULATE OBJECT INTENSITY
    intensity = np.max(obj)

    #ADD TO GRAND_DICTIONARY
    obj_name = namelist[i]
    i += 1
    grand_dict[obj_name] = [intensity]

    #ADD CENTER OF MASS TO GRAND DICT VALUE
    M = cv2.moments(contour)
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])
    com = [cx, cy]
    grand_dict[obj_name].append(com)

    #ADD TOTAL PIX AREA TO GRAND DICT VALUE
    rect = cv2.minAreaRect(contour)
    box = cv2.boxPoints(rect)
    box = np.intp(box)
```

```python
    area = int(rect[1][0] * rect[1][1])
    grand_dict[obj_name].append(area)

    #IMAGE TESTER LINE - GENERATE PIC WITH MBR BOXES
    # cv2.drawContours(image, [box], 0, (255, 0, 0), 2)

    #ADD L/R CORNERS + DIAG OF BOX TO GRAND DICT
    x_coords = [box[i][0] for i in range(4)]
    y_coords = [box[i][1] for i in range(4)]
    x_min = min(x_coords)
    y_min = min(y_coords)
    x_max = max(x_coords)
    y_max = max(y_coords)
    diag = int(np.sqrt((x_max - x_min)**2 + (y_max - y_min)**2))
    lowLeft = (x_min, y_min)
    upRight = (x_max, y_max)
    corners = [upRight, lowLeft]
    grand_dict[obj_name].append(corners)
    grand_dict[obj_name].append(diag)

    #CLASSIFY BY SIZE, ASPECT RATIO, AND SHAPE
    what = []
    size = get_box_size(grand_dict[obj_name][2])
    what.append(size)
    aspect = aspect_ratio(grand_dict[obj_name][3])
    what.append(aspect)
    shape = shape_category(contour, image)
    what.append(shape)
    grand_dict[obj_name].append(what)

    #CLASSIFY BY LOCATION AND ORIENTATION
    where = []
    vert = get_vert_section(image, contour)
    where.append(vert)
    hoz = get_hoz_section(image, contour)
    where.append(hoz)
    ornt = get_orientation(contour)
    where.append(ornt)
    grand_dict[obj_name].append(where)

    grand_dict[obj_name].append(contour)

# CREATE NEIGHBORS ARRAY FOR EACH KEY IN GRAND_DICT
for obj_name in grand_dict:
    neighbors = []
    for other_name in grand_dict:
        if other_name != obj_name:
            if check_overlap(grand_dict[other_name][3], grand_dict[obj_name][3]):
                neighbors.append(other_name)
    grand_dict[obj_name].append(neighbors)

get_twins(5) # adds geometric twins to the end of the value array
get_twins(6) # adds location twins to the end of the value array

#  CREATE NEARNESS ARRAY FOR EACH KEY IN GRAND_DICT
for obj_name in grand_dict:
    nearness = []
    for other_name in grand_dict:
        if other_name != obj_name:
            near = calculate_nearness(grand_dict[other_name][7], grand_dict[obj_name][7])
            if obj_name == "AlmaMater":
                if near < 0.3:
                    nearness.append(other_name)
            if near < 0.165:
                nearness.append(other_name)
    grand_dict[obj_name].append(nearness)

with open("output.html", "w") as f:
    for obj_name in grand_dict:
        arr1 = grand_dict[obj_name][5]
        arr2 = grand_dict[obj_name][6]
        arr3 = grand_dict[obj_name][8]
        arr4 = grand_dict[obj_name][10]
        arr5 = grand_dict[obj_name][11]
        building = obj_name
        output = building_near(building, arr1, arr2, arr3, arr4, arr5)
        output = output.replace(building, f"<b>{building}</b>")
        f.write(f"{output}<br>")

##################################################################################################
##################################################################################################

# STEP 1 DELIVERABLES GENERATOR
data_list = []
for key in grand_dict:
    data_dict = {
        "intensity": grand_dict[key][0],
        "name": key,
        "(x,y) of CoM": grand_dict[key][1],
        "pixel area": grand_dict[key][2],
        "L/R corners": grand_dict[key][3],
        "MBR Diag": grand_dict[key][4],
        "Intersectors": grand_dict[key][8]
    }
    data_list.append(data_dict)
df = pd.DataFrame(data_list)
html = df.to_html(index=False, bold_rows=True)
with open("step1.html", "w") as f:
    f.write(html)

# STEP 2 DELIVERABLES GENERATOR
data_list = []
for key in grand_dict:
    data_dict = {
        "name": key,
        "size": grand_dict[key][5][0],
        "aspect ratio": grand_dict[key][5][1],
        "shape": grand_dict[key][5][2],
        "confusion": grand_dict[key][9]
    }
    data_list.append(data_dict)
df = pd.DataFrame(data_list)
html = df.to_html(index=False, bold_rows=True)
with open("step2.html", "w") as f:
    f.write(html)

# STEP 3 DELIVERABLES GENERATOR
data_list = []
for key in grand_dict:
    data_dict = {
        "name": key,
        "verticality": grand_dict[key][6][0],
        "horizontality": grand_dict[key][6][1],
        "orientation": grand_dict[key][6][2],
        "confusion": grand_dict[key][10]
    }
    data_list.append(data_dict)
df = pd.DataFrame(data_list)
html = df.to_html(index=False, bold_rows=True)
with open("step3.html", "w") as f:
    f.write(html)

# STEP 4 DELIVERABLES GENERATOR
data_list = []
for key in grand_dict:
    data_dict = {
        "name": key,
        "near to": grand_dict[key][11],
    }
    data_list.append(data_dict)
df = pd.DataFrame(data_list)
html = df.to_html(index=False, bold_rows=True)
with open("step4.html", "w") as f:
    f.write(html)
```