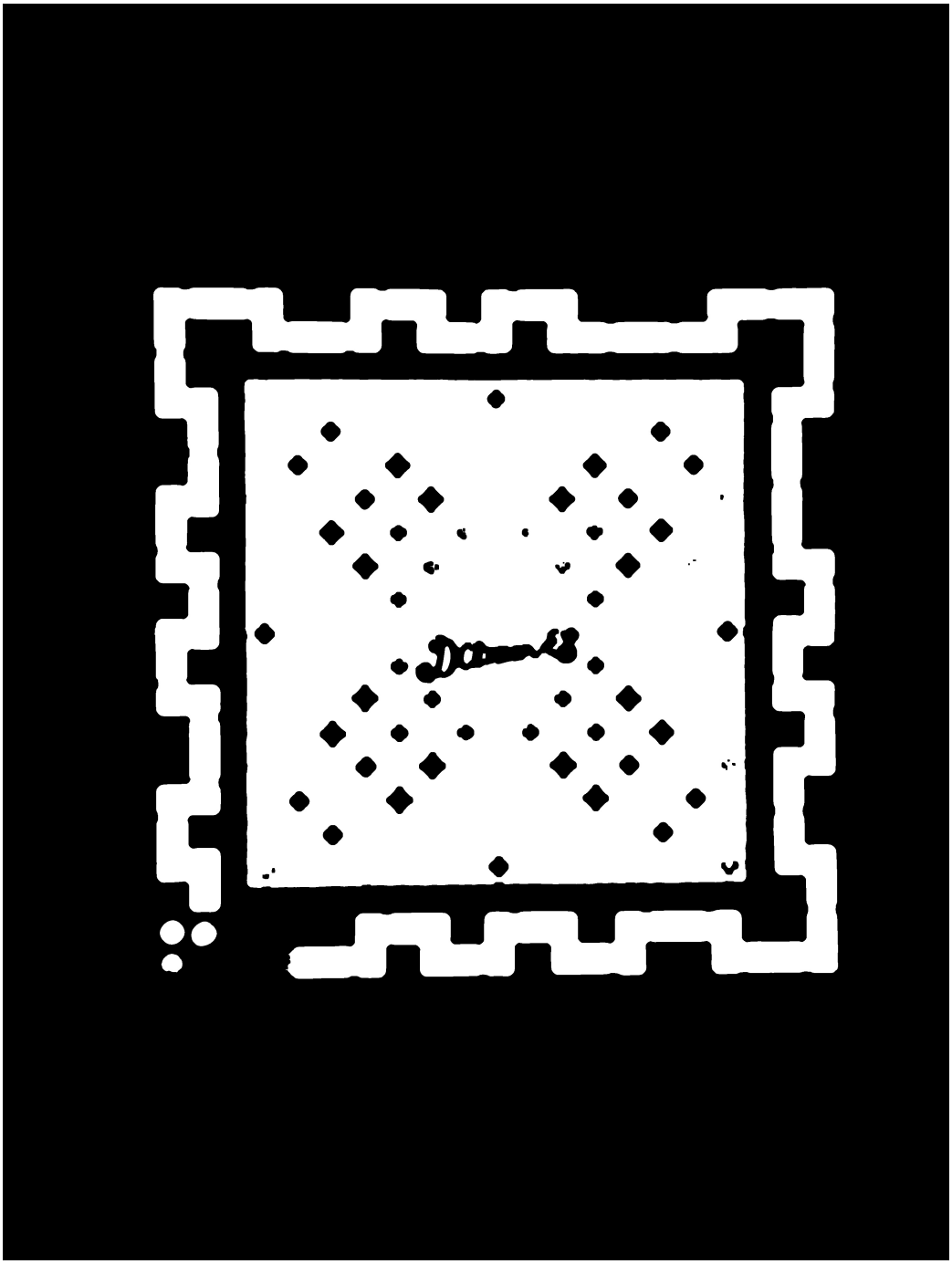
Automatic Score Calculator for the Double Dominoes Game

Board Extraction

To extract the board, I used an HSV mask to eliminate unwanted elements.
Lower limit=(85,0,0)
Upper limit=(108,0,0)
The following operations were applied to the resulting image:

```
image_gray = cv.cvtColor(res,cv.COLOR_BGR2GRAY)
#show_image("gray",image_gray)

image_m_blur = cv.medianBlur(image_gray,27)
image_g_blur = cv.GaussianBlur(image_m_blur, (0, 0), 9)
image_sharpened = cv.addWeighted(image_m_blur, 2, image_g_blur, -0, 0)
#show_image('image_sharpened',image_sharpened)

_, thresh = cv.threshold(image_sharpened, 10, 255, cv.THRESH_BINARY)

#show_image('image_thresholded',thresh)

kernel = np.ones((5, 5), np.uint8)
thresh = cv.erode(thresh, kernel)
```

Result:

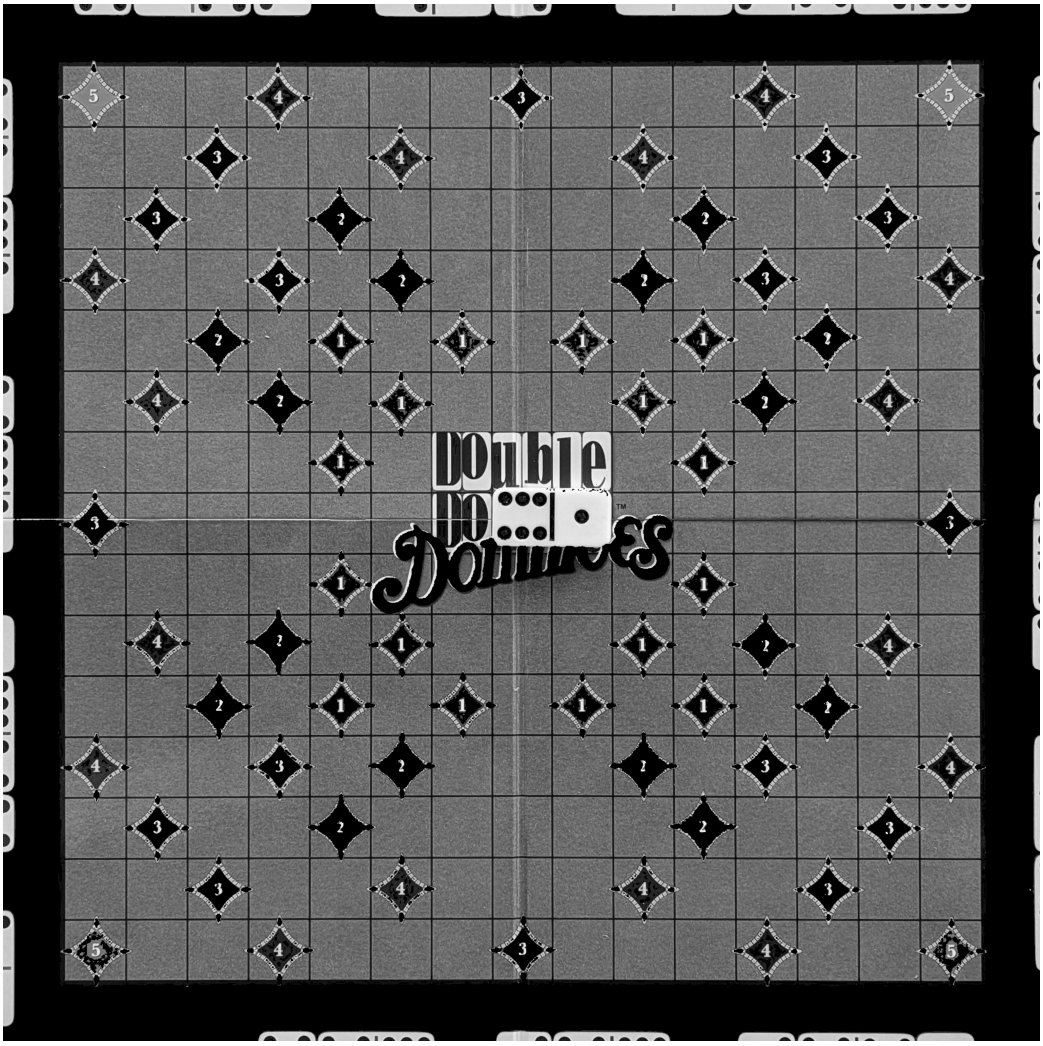


I used the Canny function from OpenCV with a lower limit of 100 and an upper limit of 175 to extract edges. Then, I found the corners for the contour

with the maximum area, forming a square (the board). I added a parameter to the board extraction function to determine whether to add an additional 2 rows and columns outside the board to recognize pieces that slightly exceed the board.
Algorithm for corner detection:

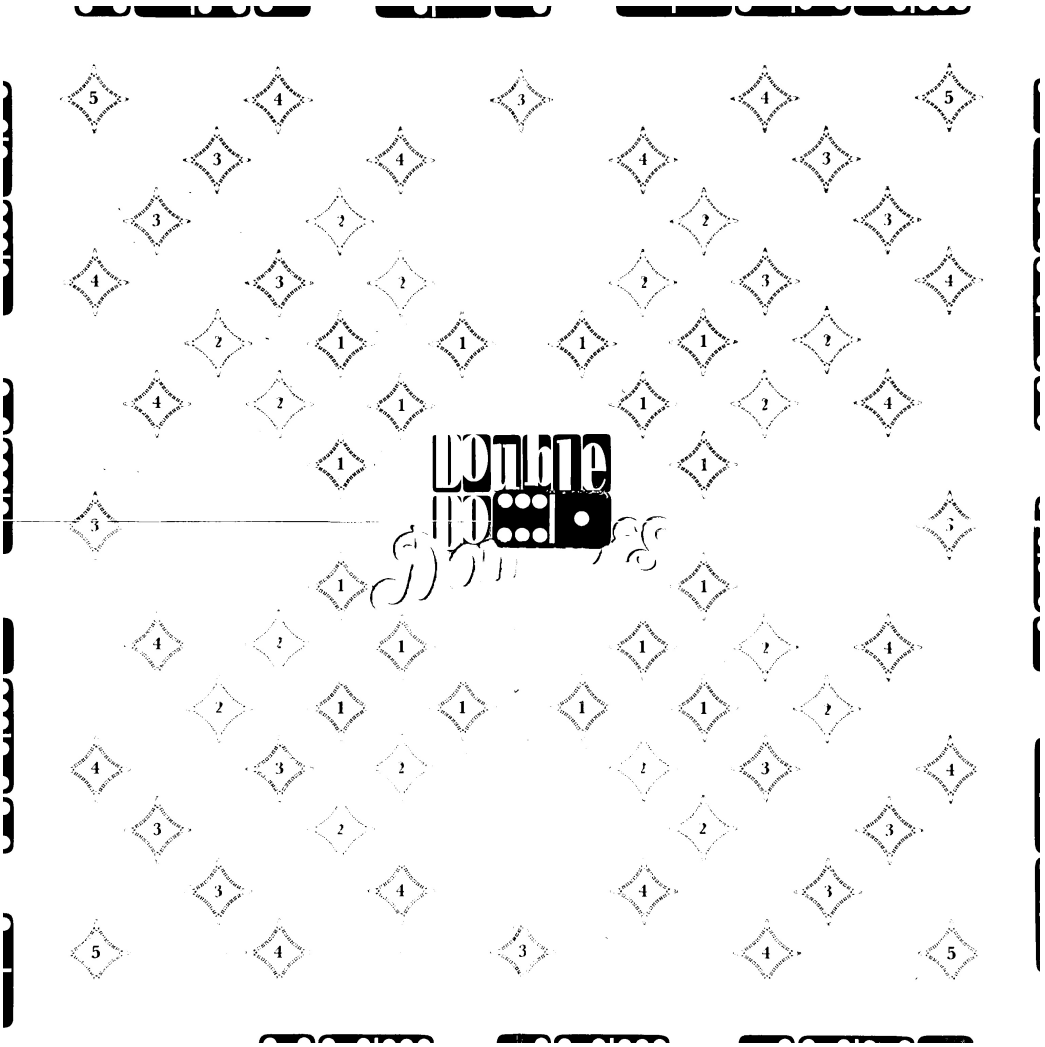
```
max_area=0
for i in range(len(contours)):
    if(len(contours[i]) >3):
        possible_top_left = None
        possible_bottom_right = None
        for point in contours[i].squeeze():
            if possible_top_left is None or point[0] + point[1] < possible_top_left[0] + possible_top_left[1]:
                possible_top_left = point

            if possible_bottom_right is None or point[0] + point[1] > possible_bottom_right[0] + possible_bottom_right[1] :
                possible_bottom_right = point

        diff = np.diff(contours[i].squeeze(), axis = 1)
        possible_top_right = contours[i].squeeze()[np.argmin(diff)]
        possible_bottom_left = contours[i].squeeze()[np.argmax(diff)]

        current_area = cv.contourArea(np.array([[possible_top_left], [possible_top_right], [possible_bottom_right], [possible_bottom_left]]))


        width = np.linalg.norm(possible_top_right - possible_top_left)
        height = np.linalg.norm(possible_bottom_left - possible_top_left)
        aspect_ratio = width / height if height != 0 else 0

        if current_area > max_area and abs(1 - aspect_ratio) < 0.1:
            max_area = cv.contourArea(np.array([[possible_top_left],[possible_top_right],[possible_bottom_right],[possible_bottom_left]]))
            top_left = possible_top_left
            bottom_right = possible_bottom_right
            top_right = possible_top_right
            bottom_left = possible_bottom_left
```

I corrected the image using the WarpPerspective function. (the image will have a width and height of 2700 pixels or 3060 pixels with additional rows and columns)
Result:



Piece Positions

I stored the rows and columns as coordinates of their start and end points. Each cell has a size of 180x180 pixels, including those outside the board.
I applied the following operation to the extracted board image to obtain a black-and-white image.
_, thresh = cv.threshold(result, 190, 255, cv.THRESH_BINARY_INV)



I initialized a 15x15 matrix with -1. In each move, I traversed it to see if the free cells corresponding to the current position have an average color less than 140. (When making this check, I added 10 pixels in each direction to this patch to avoid issues with the TemplateMatching operation described in the following sentences). Then I checked that there is no cell from the text "Double." I performed this check using the TemplateMatching function. Then, the values in the matrix corresponding to the position on the game board took the values of the numbers identified in part 2.

Piece Identification

I recorded the row and column numbers for the newly added piece and applied template matching on that piece. I added 30 pixels in all directions to make sure I have the entire piece in the image.
To create templates, I used the following approach:
I used auxiliary image 2 from which I took each piece plus 40 pixels in each direction.
I applied an HSV mask to these patches with low=(0,0,160) and high=(255,255,255) and converted them to grayscale.
I found the corners of the domino in the image using Canny (120,175) and findContours and applied WarpPerspective to straighten it.
(dimensions were width 175 and height 350).

I rotated the obtained image to have all possible variations of that particular domino.
Not all dominos turned out well due to the background. For these, I extracted them manually from Paint and applied the same HSV mask.

Scoring Calculation

I manually filled the game matrix with the places where points are received. I did the same with the pawn's path.
I stored the positions of the pawns on the path in 2 variables, and at each move, I checked if one end of the domino is equal to the number on the current position of one of the pawns and if I placed a domino on a scoring

cell. I added the obtained score to the current position. The returned result is the current score minus the calculated score up to the current round.