CSE464 - Digital Image Processing Project Final Report Optical Mark Reader

Adem Murat Özdemir - İbrahim Barış Uysal

December 28, 2024

Abstract

In this project, an Optical Mark Reader (OMR) system was developed and implemented as part of the Digital Image Processing course. The primary objective was to automate the evaluation of optically marked forms, commonly used in examinations and surveys. The system was designed to detect, process, and analyze marked responses from scanned images with high accuracy. Techniques such as image preprocessing, thresholding, contour detection, and region-of-interest segmentation were utilized to achieve reliable results. The effectiveness of the system was validated using a dataset of test forms, and performance metrics, including accuracy, were analyzed. The results demonstrated the feasibility of the proposed approach for practical applications in educational and administrative settings.

1 Introduction

Optical Mark Reader (OMR) technology has been widely employed in various fields, including education and administrative processes, due to its efficiency in automating the evaluation of optically marked forms. These systems are capable of detecting and analyzing marked responses from printed documents, thereby eliminating manual effort and reducing errors associated with human interpretation. The increasing demand for accurate and time-efficient data processing has highlighted the need for robust OMR systems. Advances in digital image processing techniques have further enhanced the capability of these systems to handle diverse form designs and varying image qualities. In this project, an OMR system was developed to address the challenge of automated evaluation using digital image processing methods. The focus was placed on detecting marked responses through a series of steps, including image acquisition, preprocessing, segmentation, and recognition. By leveraging these techniques, the system was designed to achieve high reliability and accuracy in identifying and interpreting user responses. The implementation of this project serves not only as an academic exercise in applying image processing principles but also as a practical solution to streamline the analysis of optically marked forms. The following sections present the methodology, experimental results, and conclusions derived from the project, emphasizing the potential applications of the developed system.

2 Methodology

This section presents the theoretical foundation and algorithmic approaches employed in the development of the Optical Mark Reader (OMR) system. The methodology encompasses multiple stages of image processing, each designed to contribute to robust and accurate mark detection.

2.1 Image Preprocessing

The preprocessing stage is crucial for ensuring consistent and reliable mark detection across various input conditions.

2.1.1 Color Space Conversion

Input images are first converted from RGB to grayscale using the luminosity method, which accounts for human perception of color:

$$I_{\text{gray}} = 0.299R + 0.587G + 0.114B \tag{1}$$

This weighted sum better preserves perceived brightness differences compared to simple averaging.

2.1.2 Gaussian Smoothing

To reduce noise while preserving essential features, a Gaussian filter is applied. The 2D Gaussian function used is:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

where determines the degree of smoothing. The kernel size is chosen to balance noise reduction with feature preservation.

2.2 Adaptive Thresholding

2.2.1 Local Threshold Computation

For each pixel position (x,y), the threshold is computed using:

$$T(x, y) = (x, y)C \tag{3}$$

where (x,y) is the mean of the neighborhood and C is a constant offset. Two methods are supported:

1. Mean-based threshold:

$$_{mean}(x, y) = \frac{1}{w^2} \sum_{i,jW} I(i, j) \tag{4}$$

2. Gaussian-weighted threshold:

$$_{gaussian}(x, y) = \sum_{i,jW} w_{ij} I(i, j) \tag{5}$$

where W is the neighborhood window and w$_{ij} are Gaussian weights.$

2.3 Edge Detection

The system implements the Canny edge detection algorithm with several enhancements for OMR-specific requirements.

2.3.1 Gradient Computation

Gradients are computed using Sobel operators:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad M_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{6}$$

The gradient magnitude and direction are calculated as:

$$|\nabla I| = \sqrt{(\nabla_x I)^2 + (\nabla_y I)^2}, \quad \theta = \arctan 2(\nabla_y I, \nabla_x I) \tag{7}$$

2.3.2 Non-maximum Suppression

Edge thinning is performed through non-maximum suppression using the gradient direction. The algorithm quantizes the gradient direction into four sectors and compares each pixel with its neighbors along the gradient direction.

2.3.3 Double Thresholding

Two threshold values are used to classify edge pixels:

$$E(x, y) = \begin{cases} \text{strong}, & \text{if } |\nabla I(x, y)| > T_{\text{high}} \\ \text{weak}, & \text{if } T_{\text{low}} \leq |\nabla I(x, y)| \leq T_{\text{high}} \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

2.4 Contour Detection and Analysis

The contour detection algorithm uses border following with 8-connectivity.

2.4.1 Border Following

The algorithm traces object boundaries using an 8-directional chain code:

$$D = \{(-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1)\} \tag{9}$$

2.4.2 Contour Filtering

Contours are filtered based on geometric properties:

$$\text{Valid}(C) = \begin{cases} 1, & \text{if } A_{\min} \leq \text{Area}(C) \leq A_{\max} \text{ and } R_{\min} \leq \text{AspectRatio}(C) \leq R_{\max} \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

2.5 Mark Detection

The final stage involves analyzing potential mark regions using intensity analysis.

2.5.1 Region Analysis

For each potential mark region, the average intensity is computed:

$$I_{\text{avg}} = \frac{1}{N} \sum_{(x,y)\in R} I(x, y) \tag{11}$$

2.5.2 Mark Classification

The final mark detection uses a threshold-based classifier:

$$\text{Mark}(R) = \begin{cases} 1, & \text{if } I_{\text{avg}} < T_{\text{mark}} \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

2.6 Clustering for Multiple Choice Detection

For multiple choice questions, a clustering approach is used to group related marks.

2.6.1 Spatial Clustering

Marks are clustered based on spatial proximity using a modified k-means algorithm:

$$D(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{13}$$

2.6.2 Answer Selection

The final answer for each question is determined by:

$$\text{Answer} =_{i\in\text{Options}} I_{\text{avg}}(R_i) \tag{14}$$

This comprehensive methodology ensures robust and accurate OMR processing across various input conditions while maintaining computational efficiency.

3 Implementation

3.1 Adaptive Thresholding

A custom implementation of adaptive thresholding was developed to separate foreground (marked areas) from the background while handling variations in illumination. The algorithm supports both mean and Gaussian adaptive methods, with configurable block size and constant offset parameters.

For the mean adaptive method, the threshold value $T(x, y)$ for each pixel is computed as:

$$T(x, y) = \text{mean}(I_{\text{neighborhood}}) - C \tag{15}$$

For the Gaussian adaptive method, a weighted sum is used:

$$T(x,y) = \sum_{i,j \in N} w_{ij} I_{ij} - C \tag{16}$$

where the Gaussian weights w_{ij} are computed as:

$$w_{ij} = \exp\left(-\frac{(i-i_c)^2 + (j-j_c)^2}{2(\text{block\_size}/6)^2}\right) \tag{17}$$

The implementation includes several key components:

- Edge handling using reflection padding:

$$I_{\text{padded}} = \text{pad}(I, \text{pad\_size}, \text{mode='reflect'}) \tag{18}$$

- Binary thresholding operation:

$$O(x,y) = \begin{cases} \text{max\_value}, & \text{if } I(x,y) > T(x,y) \text{ for binary} \\ 0, & \text{if } I(x,y) \leq T(x,y) \text{ for binary} \\ 0, & \text{if } I(x,y) > T(x,y) \text{ for binary\_inv} \\ \text{max\_value}, & \text{if } I(x,y) \leq T(x,y) \text{ for binary\_inv} \end{cases} \tag{19}$$

The algorithm processes each pixel with a neighborhood block defined by block_size (which must be odd and 3). The process includes:

1. Computing local statistics within each block
2. Applying the threshold operation based on the adaptive method
3. Generating the binary output image

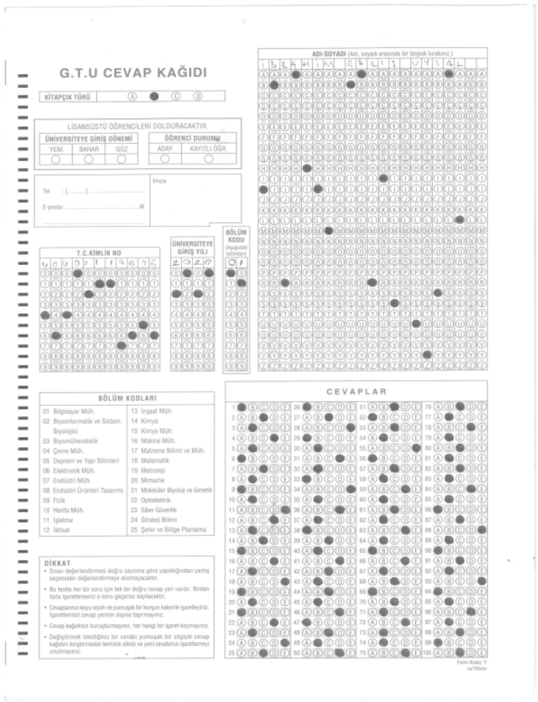
Progress tracking is implemented using tqdm for monitoring the thresholding process:

$$\text{Progress} = \frac{\text{processed\_pixels}}{\text{total\_pixels}} \times 100\% \tag{20}$$

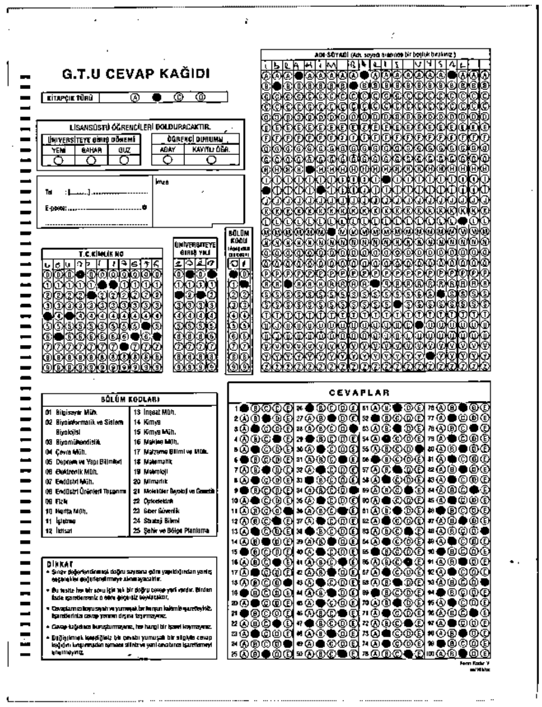
The implementation provides control over key parameters:

- max_value: Maximum value for thresholding (typically 255)
- adaptive_method: Choice between ”mean” and ”gaussian”
- threshold_type: ”binary” or ”binary_inv”
- block_size: Size of pixel neighborhood
- C: Constant subtracted from mean or weighted mean

Figure 1 shows the results of our adaptive thresholding implementation. As shown in Figure 1, the adaptive thresholding successfully separates the foreground marks and form structure from the background, handling the varying illumination conditions present in the input image.



(a) Input grayscale image



(b) Result after adaptive thresholding

Figure 1: Results of adaptive thresholding implementation showing (a) original grayscale input image and (b) binary output after applying adaptive thresholding with block size 11 and C=6

3.2 Edge Detection

A custom implementation of the Canny Edge Detection algorithm described by Canny [2] was developed to identify the boundaries of relevant structures. The process involves several key steps, implemented using NumPy arrays for efficient computation.

The first step applies Gaussian smoothing using a custom kernel:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{21}$$

where σ controls the smoothing strength and kernel size is determined by the input parameter.

For gradient estimation, the algorithm supports three filter types (Sobel, Prewitt, and Robert). Using Sobel filters as the default option:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad M_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{22}$$

The gradient magnitude and direction are computed as:

$$|G| = \sqrt{G_x^2 + G_y^2}, \quad \theta = \arctan 2(G_y, G_x) \tag{23}$$

where G_x and G_y are obtained by convolving the image with M_x and M_y respectively.

Non-maximum suppression is applied by comparing each pixel's magnitude with its neighbors along the gradient direction. The gradient angles are quantized into four directions (0°, 45°, 90°, 135°). The algorithm then uses double thresholding with two parameters:

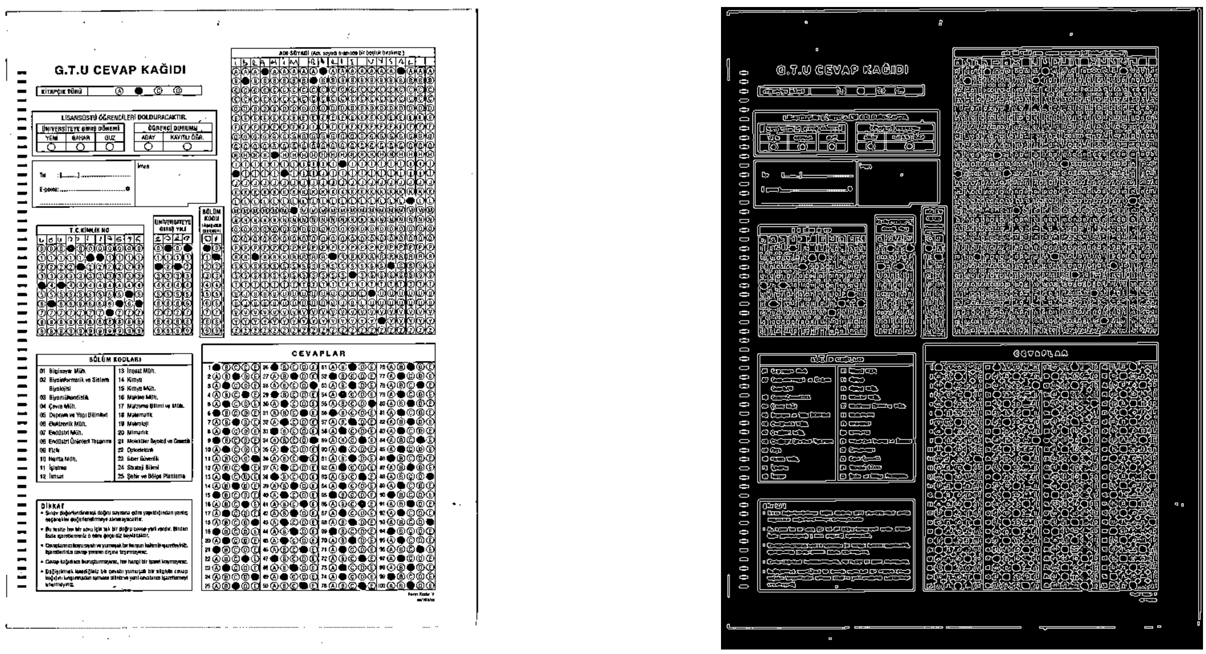
$$T_{\text{high}} = \max(|G|) \times \text{high\_ratio}, \quad T_{\text{low}} = T_{\text{high}} \times \text{low\_ratio} \tag{24}$$

Finally, hysteresis edge tracking is performed to connect edge segments:

- Strong edges (pixels $> T_{\text{high}}$) are immediately accepted
- Weak edges ($T_{\text{low}} \leq$ pixels $\leq T_{\text{high}}$) are accepted if connected to strong edges
- Remaining pixels are suppressed

The implementation includes progress tracking using tqdm library, providing real-time feedback during the edge detection process. The algorithm accepts parameters for kernel size (default=5), low threshold ratio (default=0.05), and high threshold ratio (default=0.20), allowing for fine-tuning of edge detection sensitivity.

The results of our edge detection implementation are shown in Figure 2.



(a) Input binary image after adaptive thresholding

(b) Detected edges after Canny edge detection

Figure 2: Results of Canny edge detection: (a) Binary input image from adaptive thresholding stage (b) Final edge detection output using kernel size=5, low ratio=0.05, and high ratio=0.20

The edge detection successfully identifies form structures and marked regions while suppressing noise, as shown in Figure 2. The combination of non-maximum suppression and hysteresis thresholding produces clean, continuous edges suitable for subsequent contour detection.

3.3 Custom Contour Detection Implementation

The contour detection algorithm implemented in this project is based on the border following method described by Suzuki and Abe [1]. The algorithm processes binary images to identify and trace object boundaries using 8-connectivity.

The main contour detection process relies on identifying and following object boundaries in the binary image. The algorithm maintains several key data structures including a binary image matrix, a visited pixel matrix to track processed points, a contour list storing boundary coordinates, and a hierarchy array tracking contour relationships.

The border following process uses 8-directional neighborhood scanning. The eight directions around each pixel are mathematically defined as:

$$D = \{(-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1)\} \tag{25}$$

For efficient boundary tracing, the algorithm employs a border following function that can be expressed mathematically as:

$$\text{follow\_border}(x, y) = \{(x_i, y_i) | (x_i, y_i) \text{ is connected to } (x_{i-1}, y_{i-1}) \text{ via 8-connectivity}\} \tag{26}$$

To handle image boundaries appropriately, the algorithm includes boundary validation:

$$\text{valid}(x, y) = \begin{cases} \text{True}, & \text{if } 0 \leq x < \text{width and } 0 \leq y < \text{height} \\ \text{False}, & \text{otherwise} \end{cases} \tag{27}$$

When following a border, the algorithm: 1. Marks the current pixel as visited 2. Examines neighboring pixels in clockwise order 3. Updates the direction based on the previous movement 4. Continues until returning to the start point or finding no more border pixels

The implementation offers several advantages including simple implementation using pure Python/NumPy and direct access to contour points without approximation. However, it also has limitations such as being less optimized compared to OpenCV's native implementation, using a basic hierarchy structure without complex topological analysis, and lacking built-in contour approximation or filtering capabilities.

The resulting contours are represented as a set:

$$\text{Contours} = \{C_1, C_2, ..., C_n\} \text{ where } C_i = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\} \tag{28}$$

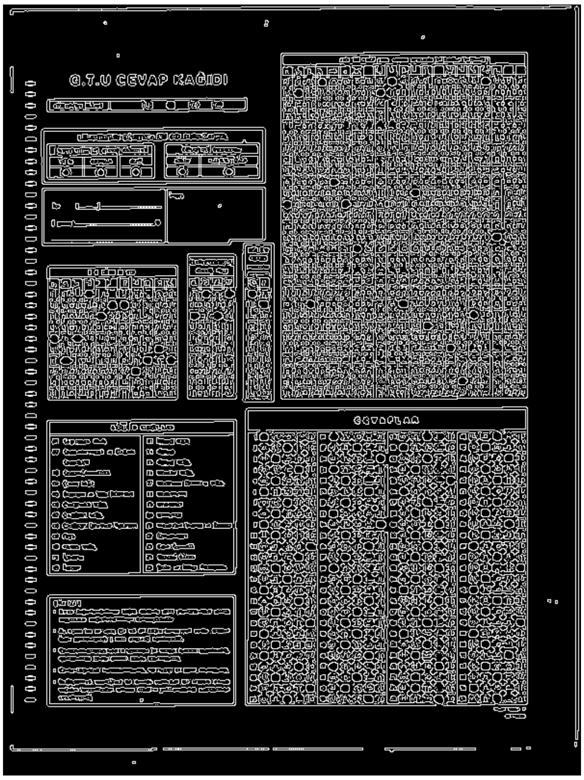
where each contour C_i represents a sequence of connected boundary points in the image.

3.4 Mark Detection

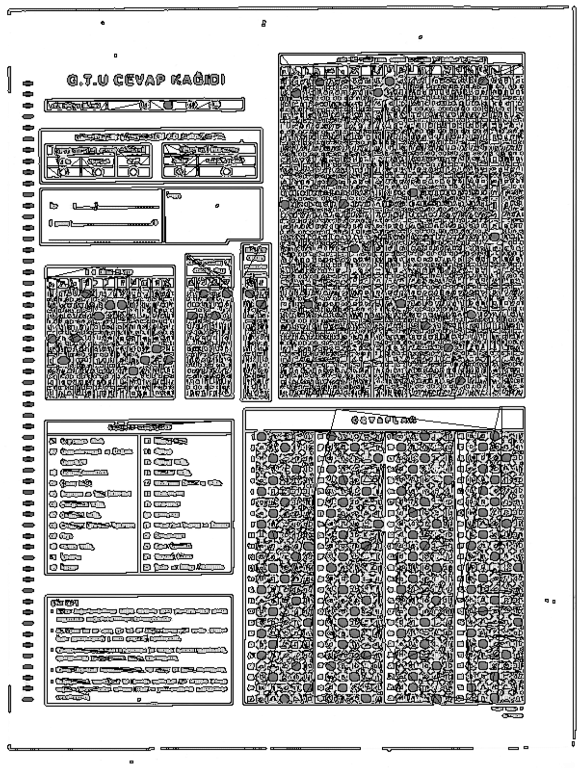
The system employs a multi-stage approach for mark detection, combining coordinate transformation, morphological operations, clustering, and intensity analysis.

3.4.1 Coordinate Transformation

The mark detection process begins by reading a CSV file named "answers.csv" which contains reference pixel coordinates for the answer sheet. These coordinates are transformed



(a) Input binary edge image



(b) Detected contours overlaid on original image

Figure 3: Comparison of contour detection results. (a) Shows the input binary edge image after preprocessing. (b) Demonstrates the detected contours using our custom implementation based on the Suzuki-Abe algorithm. The contours are shown in different colors to distinguish between separate objects and hierarchical relationships.

from the original answer sheet rectangle to the detected question rectangle in the scanned image using a perspective transformation matrix computed by the `cv2.getPerspectiveTransform` function:

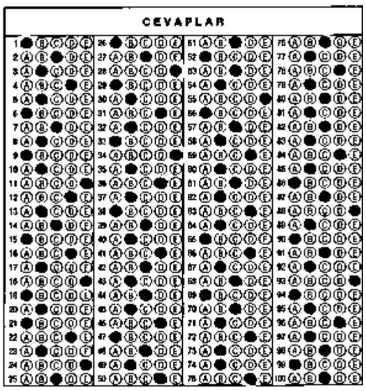
$$\text{Transformed}_{\text{answers}} = \text{map\_answers\_to\_new\_rectangle}(\text{original\_corners}, \text{new\_corners}, \text{answers}) \tag{29}$$

where `original_corners` and `new_corners` represent the corners of the original and detected rectangles, respectively.

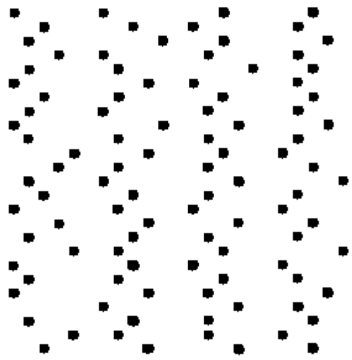
3.4.2 Y-Coordinate Optimization

To enhance the regions where marks are expected to be found, morphological opening operations are applied to the binary image of the question area:

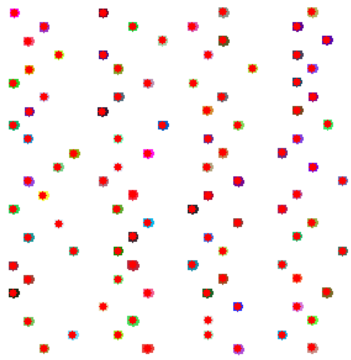
$$R_{\text{processed}} = \text{morph\_open}(\text{morph\_ellipse}(R)) \tag{30}$$



(a) Input Questions Box



(b) Morphological Opening Operations Result



(c) Clustering Method (Centroid) Output

Figure 4: Clustering Method Output: (a) Original questions box found in the binary image (b) Result after applying morphological opening operations (c) Visualization of the found marks with the clustering method

Before identifying potential mark regions, the system first locates reference rectangles on the answer sheet. These rectangles serve as a guide for updating the Y-coordinates of the potential mark centers. The reference rectangles are detected using contour analysis and filtered based on their aspect ratio and position on the sheet. The `find_rectangle_groups` function is used to identify groups of rectangles with similar X-coordinates. The function checks for a group containing exactly 57 rectangles, which corresponds to the expected number of reference rectangles on the answer sheet. If a group with 57 rectangles is found, it is considered as the set of reference rectangles. Once the reference rectangles are identified, their Y-coordinates are used to update the Y-coordinates of the potential mark centers. The `find_centers_clustering` function is applied to the binary image of the question area to identify potential mark regions using connected component analysis and clustering , as shown in Figure 4. The function returns the initial Y-coordinates for each potential mark center. To refine the Y-coordinates, the system matches each potential mark center with its corresponding reference rectangle based on their vertical proximity. The Y-coordinate of each potential mark center is then updated to align with the center of its corresponding reference rectangle. The updated Y-coordinates are calculated as follows:

$$y_{\text{updated}} = y_{\text{ref}} + \frac{h_{\text{ref}}}{2} \tag{31}$$

where y_{updated} is the updated Y-coordinate of the potential mark center, y_{ref} is the Y-coordinate of the corresponding reference rectangle, and h_{ref} is the height of the reference rectangle. By updating the Y-coordinates based on the reference rectangles, the system ensures accurate vertical alignment of the potential mark centers, even in the presence of slight variations or distortions in the scanned image.

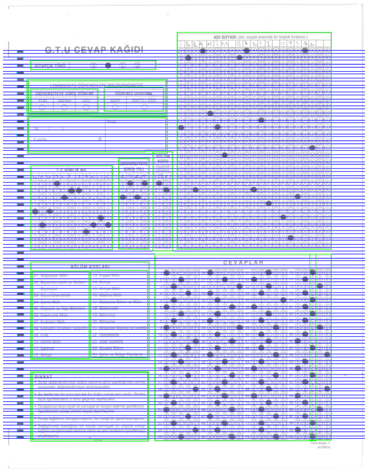


Figure 5: Y-axis Updated reference rectangles with clustering method output

3.4.3 Intensity Analysis

For each potential mark position (both name/surname and answers):

1. A region of interest (ROI) is defined around the final coordinates: - Centered at (x,y) from previous step - ROI dimensions: ±4 pixels horizontally, full reference rectangle height vertically
2. Average intensity calculation within ROI:

$$I_{\text{mark}} = \frac{1}{N} \sum_{(x,y)\in \text{ROI}} I(x, y) \tag{32}$$

3. For answer fields, the marked option is determined by:

$$\text{Marked}_{\text{option}} =_{o\in\{a,b,c,d,e\}} I_{\text{mark}}(o) \tag{33}$$

This approach ensures accurate mark detection by:

Maintaining precise X-coordinate positions from reference data. Using clustering to optimize Y-coordinates for all fields. Analyzing intensity values in well-defined regions around each point, as shown in the boxes used for intensity calculations in Figure 6.

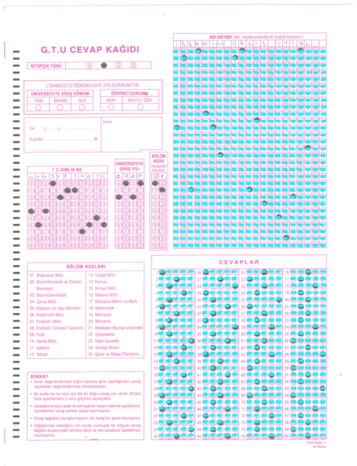


Figure 6: Boxes that used in intensity calculations

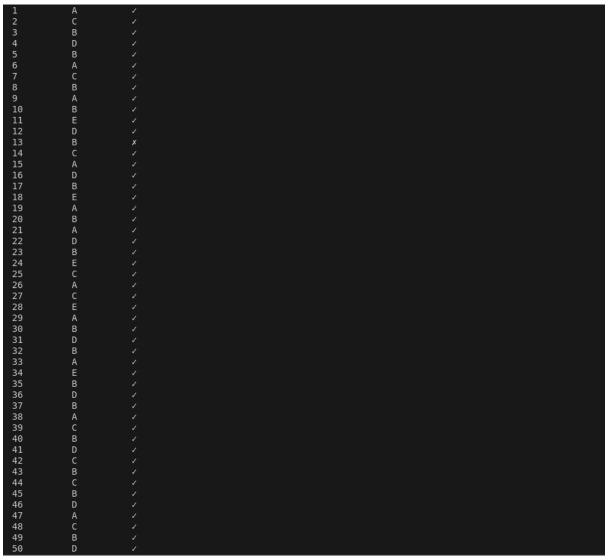
4 Experiment

The performance of the Optical Mark Reader (OMR) system was evaluated through four different trials using a single scanned optical form as input. The form contained an answers section with 100 questions, where marked responses were analyzed for accuracy. Since the input image was directly obtained from a scanned source, it provided ideal conditions with no distortions or imperfections, allowing the system to operate with high precision. The detection of marked circles was based on intensity measurement. In trials where the circles were fully darkened, the system achieved 100% accuracy in detecting and interpreting the marked responses. However, in cases where the circles were partially filled or not sufficiently dark, the accuracy rate decreased slightly, highlighting the system's sensitivity to the intensity levels of the marked responses.The use of a single high-quality scanned form with 100 answer fields and reference rectangles, along with multiple trials, highlights the controlled nature of this experiment. These results demonstrate the system's effectiveness under ideal conditions and its limitations when dealing with varying response intensities. Future evaluations may explore methods to improve the robustness of marked response detection under less ideal conditions.

5 Challenges Encountered

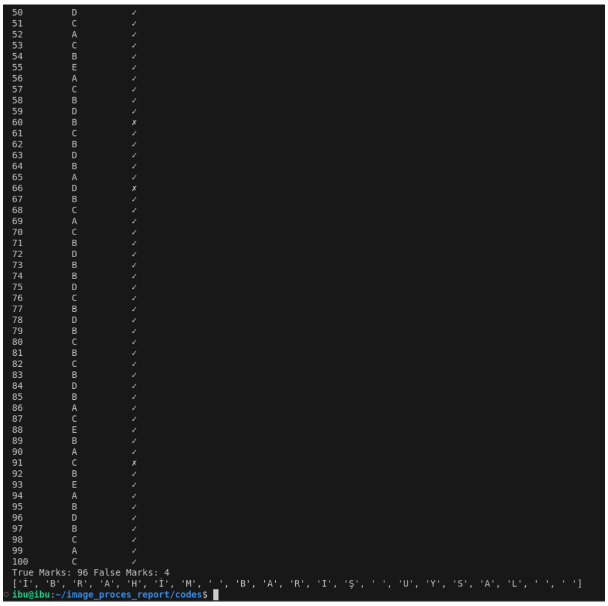
An initial approach involved applying perspective transformation to correct skewed images and align them accurately. However, the results were highly unsatisfactory, as significant distortions occurred in the transformed image. These distortions affected the geometry of the form, causing misalignment of critical elements such as the reference rectangles on the left side. After extensive attempts to improve the results, this approach was ultimately abandoned, as it proved unsuitable for maintaining the form's structural integrity. A significant amount of time was spent on this issue, which delayed other parts of the development process.

6 Final Result



```
1         A         ✓
2         C         ✓
3         B         ✓
4         D         ✓
5         B         ✓
6         A         ✓
7         C         ✓
8         B         ✓
9         A         ✓
10        B         ✓
11        E         ✓
12        D         ✓
13        B         ✗
14        C         ✓
15        A         ✓
16        D         ✓
17        B         ✓
18        E         ✓
19        A         ✓
20        B         ✓
21        A         ✓
22        D         ✓
23        B         ✓
24        E         ✓
25        C         ✓
26        A         ✓
27        C         ✓
28        E         ✓
29        A         ✓
30        B         ✓
31        D         ✓
32        B         ✓
33        A         ✓
34        E         ✓
35        B         ✓
36        D         ✓
37        B         ✓
38        A         ✓
39        C         ✓
40        B         ✓
41        D         ✓
42        C         ✓
43        B         ✓
44        C         ✓
45        B         ✓
46        D         ✓
47        A         ✓
48        C         ✓
49        B         ✓
50        D         ✓
```

Figure 7: First part of output



```
50        D         ✓
51        C         ✓
52        A         ✓
53        C         ✓
54        B         ✓
55        E         ✓
56        A         ✓
57        C         ✓
58        B         ✓
59        D         ✓
60        B         ✗
61        C         ✓
62        B         ✓
63        D         ✓
64        B         ✓
65        A         ✓
66        D         ✗
67        B         ✓
68        C         ✓
69        A         ✓
70        C         ✓
71        B         ✓
72        D         ✓
73        B         ✓
74        B         ✓
75        D         ✓
76        C         ✓
77        B         ✓
78        D         ✓
79        B         ✓
80        C         ✓
81        B         ✓
82        C         ✓
83        B         ✓
84        D         ✓
85        B         ✓
86        A         ✓
87        C         ✓
88        E         ✓
89        B         ✓
90        A         ✓
91        C         ✗
92        B         ✓
93        E         ✓
94        A         ✓
95        B         ✓
96        D         ✓
97        B         ✓
98        C         ✓
99        A         ✓
100       C         ✓
True Marks: 96 False Marks: 4
['İ', 'B', 'R', 'A', 'H', 'İ', 'M', ' ', 'B', 'A', 'R', 'I', 'Ş', ' ', 'U', 'Y', 'S', 'A', 'L', ' ', ' ']
ibu@ibu:~/image_proces_report/codes$
```

Figure 8: Second part of output

7 Conclusion

The Optical Mark Reader (OMR) system demonstrated high accuracy with the test form, showcasing the potential of image processing techniques for automating the evaluation of optically marked forms. Future work will focus on addressing the challenges and evaluating the system under diverse conditions.

References

[1] Satoshi Suzuki, Keiichi Abe, *Topological structural analysis of digitized binary images by border following*, Computer Vision, Graphics, and Image Processing, Volume 30, Issue 1, 1985, Pages 32-46, ISSN 0734-189X, `https://doi.org/10.1016/0734-189X(85)90016-7`.

[2] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986, doi: 10.1109/TPAMI.1986.4767851.

Appendix

A Adaptive Treshold

```
def adaptive_threshold(image, max_value, adaptive_method,
    threshold_type, block_size, C):

if block_size % 2 == 0 or block_size < 3:
    raise ValueError("block_size␣must␣be␣an␣odd␣number␣and␣>=␣3.")

# Pad the image to handle edges
pad_size = block_size // 2
padded_image = np.pad(image, pad_size, mode='reflect')

# Output image
output = np.zeros_like(image, dtype=np.uint8)

# Pre-compute Gaussian kernel if needed
if adaptive_method == "gaussian":
    kernel = np.outer(
        np.exp(-0.5 * (np.arange(block_size) - pad_size)**2 / (
            block_size / 6)**2),
        np.exp(-0.5 * (np.arange(block_size) - pad_size)**2 / (
            block_size / 6)**2)
    )
    kernel /= kernel.sum()

# Total number of pixels to process
total_pixels = image.shape[0] * image.shape[1]

# Initialize progress bar
with tqdm(total=total_pixels, desc="Processing", unit="pixels") as
    pbar:
    # Compute the threshold for each pixel
    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            # Extract the local block
            block = padded_image[y:y + block_size, x:x + block_size]

            # Compute the threshold value based on the adaptive
                method
            if adaptive_method == "mean":
                threshold_value = block.mean()
            elif adaptive_method == "gaussian":
                threshold_value = (block * kernel).sum()
            else:
                raise ValueError("Invalid␣adaptive␣method.␣Use␣'mean'␣
```

```
                or␣'gaussian'.")

            # Apply thresholding
            if threshold_type == "binary":
                output[y, x] = max_value if image[y, x] > (
                    threshold_value - C) else 0
            elif threshold_type == "binary_inv":
                output[y, x] = 0 if image[y, x] > (threshold_value - C)
                     else max_value
            else:
                raise ValueError("Invalid␣threshold␣type.␣Use␣'binary'␣
                    or␣'binary_inv'.")

            # Update progress bar
            pbar.update(1)

    return output
```

B Canny Edge Detection

```
def apply_filtering(input_image, kernal):
    output_image = []
    kernal_size = len(kernal)
    kernal_half = kernal_size // 2
    rows_count = len(input_image)
    columns_count = len(input_image[0])

    # Show progress for padding operation
    print("Padding␣image...")
    image_copy = copy.deepcopy(input_image)

    for i in tqdm(range(rows_count), desc="Padding␣rows"):
        for j in range(kernal_half):
            image_copy[i].insert(0, input_image[i][-1-j])
            image_copy[i].append(input_image[i][j])

    for i in range(kernal_half):
        image_copy.append(image_copy[2*i])
        image_copy.insert(0, image_copy[-2-2*i].copy())

    new_rows_count = len(image_copy)
    new_columns_count = len(image_copy[0])

    print("Applying␣convolution...")
    # Show progress for convolution operation
```

```
    for i in tqdm(range(kernal_half, new_rows_count - kernal_half), desc=
        "Processing␣rows"):
        output_row = []
        for j in range(kernal_half, new_columns_count - kernal_half):
            sum = 0
            for x in range(len(kernal)):
                for y in range(len(kernal)):
                    x1 = i + x - kernal_half
                    y1 = j + y - kernal_half
                    sum += image_copy[x1][y1] * kernal[x][y]
            output_row.append(sum)
        output_image.append(output_row)

    return output_image

def get_gaussian_kernel(kernal_size, sigma=1):
    print("Generating␣Gaussian␣kernel...")
    gaussian_kernal = np.zeros((kernal_size, kernal_size), np.float32)
    size = kernal_size//2

    for x in tqdm(range(-size, size+1), desc="Computing␣kernel"):
        for y in range(-size, size+1):
            a = 1/(2*np.pi*(sigma**2))
            b = np.exp(-(x**2 + y**2)/(2* sigma**2))
            gaussian_kernal[x+size, y+size] = a*b
    return gaussian_kernal/gaussian_kernal.sum()

def gradient_estimate(image, gradient_estimation_filter_type):
    print("Estimating␣gradients...")
    if (gradient_estimation_filter_type=="sobel"):
        Mx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
        My = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)
    elif (gradient_estimation_filter_type=="prewitt"):
        Mx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], np.float32)
        My = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]], np.float32)
    else:
        Mx = np.array([[0, 1], [-1, 0]], np.float32)
        My = np.array([[1, 0], [0, -1]], np.float32)

    print("Computing␣X␣gradient...")
    X = apply_filtering(image, Mx)
    print("Computing␣Y␣gradient...")
    Y = apply_filtering(image, My)

    G = np.hypot(X, Y)
    G = G / G.max() * 255
    theta = np.arctan2(Y, X)
```

```
    return (G, theta)

def non_maxima_suppression(image, gradient_direction):
    print("Applying␣non-maximum␣suppression...")
    rows_count = len(image)
    columns_count = len(image[0])

    output_image = np.zeros((rows_count, columns_count), dtype=np.int32)
    theta = gradient_direction * 180. / np.pi
    theta[theta < 0] += 180

    for i in tqdm(range(1, rows_count-1), desc="Processing␣pixels"):
        for j in range(1, columns_count-1):
            next = 255
            previous = 255
            if (0 <= theta[i,j] < 22.5) or (157.5 <= theta[i,j] <= 180):
                next = image[i, j+1]
                previous = image[i, j-1]
            elif (22.5 <= theta[i,j] < 67.5):
                next = image[i+1, j-1]
                previous = image[i-1, j+1]
            elif (67.5 <= theta[i,j] < 112.5):
                next = image[i+1, j]
                previous = image[i-1, j]
            elif (112.5 <= theta[i,j] < 157.5):
                next = image[i-1, j-1]
                previous = image[i+1, j+1]

            if (image[i,j] >= next) and (image[i,j] >= previous):
                output_image[i,j] = image[i,j]
            else:
                output_image[i,j] = 0

    return output_image

def double_threshold(image, low_threshold_ratio, high_threshold_ratio):
    print("Applying␣double␣threshold...")
    high_threshold = image.max() * high_threshold_ratio
    low_threshold = high_threshold * low_threshold_ratio

    rows_count = len(image)
    columns_count = len(image[0])
    output_image = np.zeros((rows_count, columns_count), dtype=np.int32)

    weak = np.int32(25)
    strong = np.int32(255)

    strong_i = []
```

```
    strong_j = []
    weak_i = []
    weak_j = []

    for i in tqdm(range(len(image)), desc="Thresholding␣pixels"):
        for j in range(len(image[0])):
            if (image[i,j]>=high_threshold):
                strong_i.append(i)
                strong_j.append(j)
            if ((image[i,j] <= high_threshold) & (image[i,j] >=
               low_threshold)):
                weak_i.append(i)
                weak_j.append(j)

    strong_i = np.array(strong_i)
    strong_j = np.array(strong_j)
    weak_i = np.array(weak_i)
    weak_j = np.array(weak_j)

    output_image[strong_i, strong_j] = strong
    output_image[weak_i, weak_j] = weak

    return output_image

def hysteresis_edge_track(image):
    print("Performing␣edge␣tracking...")
    weak = np.int32(25)
    strong = np.int32(255)

    rows_count = len(image)
    columns_count = len(image[0])

    for i in tqdm(range(1, rows_count-1), desc="Tracking␣edges"):
        for j in range(1, columns_count-1):
            if (image[i,j] == weak):
                if ((image[i+1, j-1] == strong) or (image[i+1, j] ==
                    strong) or (image[i+1, j+1] == strong)
                    or (image[i, j-1] == strong) or (image[i, j+1] ==
                        strong)
                    or (image[i-1, j-1] == strong) or (image[i-1, j] ==
                        strong) or (image[i-1, j+1] == strong)):
                    image[i, j] = strong
                else:
                    image[i, j] = 0
    return image

def apply_canny_edge_detection(file_path,image1,image_name, kernal_size
   =3,
```

```
                       low_threshold_ratio=0.05,
                       high_threshold_ratio=0.09,
                       gradient_estimation_filter_type="sobel"):

start_time = time.time()
print("Starting␣Canny␣edge␣detection...")

print("\nStep␣1/6:␣Loading␣and␣converting␣image␣to␣grayscale...")
image=None
if file_path:
    image = load_image(image_name)
else:
    image=image1
# gray_scaled_image = convert_to_gray_scale(image)
gray_scaled_image=image
print("\nStep␣2/6:␣Applying␣Gaussian␣filter...")
kernal = get_gaussian_kernel(kernal_size)
image_without_noise = apply_filtering(gray_scaled_image.tolist(),
   kernal)

print("\nStep␣3/6:␣Estimating␣gradients...")
assert (gradient_estimation_filter_type in ["sobel", "prewitt", "
   robert"]), \
      "gradient␣estimation␣filter␣type␣should␣be␣[\"prewitt\",␣\"
         sobel\",␣\"robert\"]"
G, theta = gradient_estimate(image_without_noise,
   gradient_estimation_filter_type)

print("\nStep␣4/6:␣Applying␣non-maximum␣suppression...")
image_with_thin_edges = non_maxima_suppression(G, theta)
cv.imwrite("ImageAfterNon-MaximaSuppression1.jpg", np.array(
   image_with_thin_edges))

print("\nStep␣5/6:␣Applying␣double␣threshold...")
final_image = double_threshold(image_with_thin_edges,
   low_threshold_ratio, high_threshold_ratio)
cv.imwrite("ImageAfterApplyDoubleThreshold1.jpg", np.array(
   final_image))

print("\nStep␣6/6:␣Applying␣hysteresis␣edge␣tracking...")
img = hysteresis_edge_track(final_image)
cv.imwrite("FinalImage1.jpg", np.array(img))

print(f"\nCanny␣edge␣detection␣completed␣in␣{time.time()␣-␣start_time
   :.2f}␣seconds")
return final_image
```

C Clustering

```
    def find_neighbors(x, y, image):
#        """Find connected neighbors using 8-connectivity."""
    height, width = image.shape
    neighbors = []

    # Check all 8 neighboring pixels
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue

            new_x, new_y = x + dx, y + dy

            # Check boundaries
            if 0 <= new_x < width and 0 <= new_y < height:
                if image[new_y, new_x] == 0: # If pixel is black
                    neighbors.append((new_x, new_y))

    return neighbors
def connected_components(image, min_radius=2, max_radius=10):
    """
    Find connected components in binary image with size constraints.
    Returns labels and centroids.

    Parameters:
    - image: binary image
    - min_radius: minimum radius of dots to detect
    - max_radius: maximum radius of dots to detect
    """
    height, width = image.shape
    labels = np.zeros((height, width), dtype=int)
    current_label = 1
    centroids = []

    # First pass: label all pixels
    for y in range(height):
        for x in range(width):
            if image[y, x] == 0 and labels[y, x] == 0: # Unlabeled black
               pixel
                # Start a new component
                pixels_to_check = [(x, y)]
                pixel_coords = []

                # Label all connected pixels
                while pixels_to_check:
                    curr_x, curr_y = pixels_to_check.pop()
```

```
                if labels[curr_y, curr_x] == 0:
                    labels[curr_y, curr_x] = current_label
                    pixel_coords.append((curr_x, curr_y))

                    # Add neighbors to check
                    neighbors = find_neighbors(curr_x, curr_y, image)
                    pixels_to_check.extend(neighbors)

            # Calculate component size and centroid
            if pixel_coords:
                x_coords, y_coords = zip(*pixel_coords)

                # Calculate approximate radius from area
                area = len(pixel_coords)
                approx_radius = np.sqrt(area / np.pi)

                # Only keep components within radius range
                if min_radius <= approx_radius <= max_radius:
                    centroid_x = int(np.mean(x_coords))
                    centroid_y = int(np.mean(y_coords))
                    centroids.append((centroid_x, centroid_y))
                else:
                    # Reset labels for components outside size range
                    for px, py in pixel_coords:
                        labels[py, px] = 0

            current_label += 1

    return labels, centroids

def find_circle_centers(binary_image, min_radius=2, max_radius=10):
    """
    Find centers of circles in binary image.

    Parameters:
    - binary_image: input binary image
    - min_radius: minimum radius of dots to detect
    - max_radius: maximum radius of dots to detect

    Returns:
    - labels and centroids of detected dots
    """
    image = binary_image.copy()

    # Find components and their centers with size constraints
    labels, centroids = connected_components(image, min_radius,
        max_radius)
```

```
    return labels, centroids

def visualize_labels_and_centers(original_image, labels, centroids):
    """
    Visualize both labels and centroids on the image.
    Different colors for different labels and marks centroids.
    """
    # Create RGB image for visualization
    height, width = original_image.shape[:2]
    result = np.zeros((height, width, 3), dtype=np.uint8)

    # Create random colors for each label
    n_labels = len(centroids) + 1 # +1 for background
    colors = np.random.randint(0, 255, size=(n_labels, 3), dtype=np.uint8
       )
    colors[0] = [255, 255, 255] # background color = white

    # Color each component with a different color
    for y in range(height):
        for x in range(width):
            result[y, x] = colors[labels[y, x]]

    # Draw centroids as red dots
    for (x, y) in centroids:
        x, y = int(x), int(y)
        cv2.circle(result, (x, y), 3, (0, 0, 255), -1) # red dot

    return result
```

D Contour

```
def findContours(binary_image):
"""
Implements a simple version of Suzuki and Abe's algorithm for
   finding contours in binary images.

Args:
    binary_image (np.ndarray): Binary image where contours are to be
         detected.

Returns:
    contours (list of list of tuple): List of contours, where each
        contour is a list of (x, y) coordinates.
    hierarchy (np.ndarray): Information about image topology; (next,
         previous, first_child, parent).
"""
```

```
# Ensure the image is binary
binary_image = (binary_image > 0).astype(np.uint8)

height, width = binary_image.shape
contours = []
hierarchy = []

visited = np.zeros_like(binary_image, dtype=bool)

def follow_border(start_x, start_y, label):
    """Follow the border of a binary component."""
    contour = []
    directions = [(-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1),
       (0, -1), (-1, -1)] # 8-neighborhood

    x, y = start_x, start_y
    first_direction = 0

    while True:
        contour.append((x, y))
        visited[y, x] = True

        found_next = False
        for i in range(len(directions)):
            dir_idx = (first_direction + i) % len(directions)
            dx, dy = directions[dir_idx]
            nx, ny = x + dx, y + dy

            if 0 <= nx < width and 0 <= ny < height and binary_image[
               ny, nx] == 1:
                if not visited[ny, nx]:
                    x, y = nx, ny
                    first_direction = (dir_idx + 5) % len(directions) #
                         Reverse the direction
                    found_next = True
                    break

        if not found_next:
            break

        if (x, y) == (start_x, start_y):
            break

    return contour

for y in range(height):
    for x in range(width):
        if binary_image[y, x] == 1 and not visited[y, x]:
```

```
            contour = follow_border(x, y, len(contours) + 1)
            if contour:
                contours.append(contour)
                # Append dummy hierarchy data for now
                hierarchy.append([-1, -1, -1, -1])

    hierarchy = np.array(hierarchy, dtype=np.int32)
    return contours, hierarchy
```

E Helpers

```
    def create_gaussian_kernel(kernel_size=5, sigma=1.0):
    """
    Create a 2D Gaussian kernel for blurring.
    """
    # Create coordinate matrices
    ax = np.linspace(-(kernel_size - 1)/2., (kernel_size - 1)/2.,
       kernel_size)
    xx, yy = np.meshgrid(ax, ax)

    # Calculate Gaussian values
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(
       sigma))

    # Normalize the kernel
    return kernel / np.sum(kernel)
def medianBlur(image, ksize):
    # Ensure ksize is odd
    if ksize % 2 == 0:
        raise ValueError("Kernel size must be odd")

    # Get image dimensions
    if len(image.shape) == 3:
        height, width, channels = image.shape
    else:
        height, width = image.shape
        channels = 1
        image = image.reshape(height, width, 1)

    # Calculate padding size
    pad = ksize // 2

    # Create output image
    output = np.zeros_like(image)

    # Process each pixel
    for y in range(height):
```

```
        for x in range(width):
            for c in range(channels):
                # Get neighborhood
                neighborhood = []
                for i in range(-pad, pad + 1):
                    for j in range(-pad, pad + 1):
                        # Calculate neighbor coordinates
                        ny, nx = y + i, x + j

                        # Handle border by mirroring
                        if ny < 0:
                            ny = abs(ny)
                        elif ny >= height:
                            ny = 2 * height - ny - 2

                        if nx < 0:
                            nx = abs(nx)
                        elif nx >= width:
                            nx = 2 * width - nx - 2

                        neighborhood.append(image[ny, nx, c])

                # Calculate median
                output[y, x, c] = np.median(neighborhood)

    # Return same shape as input
    if channels == 1:
        output = output.reshape(height, width)

    return output.astype(np.uint8)
def gaussian_blur(image, kernel_size=5, sigma=1.0):
    """
    Apply Gaussian blur using convolution.
    """
    # Get kernel
    kernel = create_gaussian_kernel(kernel_size, sigma)

    # Pad the image
    pad_size = kernel_size // 2
    if len(image.shape) == 3:
        padded = np.pad(image, ((pad_size, pad_size), (pad_size, pad_size)
            , (0, 0)), mode='reflect')
    else:
        padded = np.pad(image, ((pad_size, pad_size), (pad_size, pad_size)
            ), mode='reflect')

    # Initialize output array
    output = np.zeros_like(image, dtype=np.float32)
```

```
    # Apply convolution
    if len(image.shape) == 3:
        for i in range(3): # For each color channel
            for y in range(image.shape[0]):
                for x in range(image.shape[1]):
                    output[y, x, i] = np.sum(
                        padded[y:y+kernel_size, x:x+kernel_size, i] *
                            kernel
                    )
    else:
        for y in range(image.shape[0]):
            for x in range(image.shape[1]):
                output[y, x] = np.sum(
                    padded[y:y+kernel_size, x:x+kernel_size] * kernel
                )

    return np.clip(output, 0, 255).astype(np.uint8)

def bgr_to_gray(image):
    """
    Convert BGR image to grayscale using weighted sum method.
    """
    # Standard weights for BGR to grayscale conversion
    weights = np.array([0.114, 0.587, 0.299])

    # Calculate weighted sum
    grayscale = np.dot(image[..., :3], weights)

    return grayscale.astype(np.uint8)

def gray_to_bgr(image):
    """
    Convert grayscale image to BGR by replicating the channel.
    """
    if len(image.shape) == 3 and image.shape[2] == 1:
        image = image[:, :, 0]

    # Convert to uint8 if not already
    if image.dtype != np.uint8:
        image = (image * 255).astype(np.uint8)

    # Create 3-channel BGR image by repeating the grayscale channel
    bgr_image = np.dstack((image, image, image))

    return bgr_image

def bounding_rect(contour):
```

```
    """
    Custom implementation of cv2.boundingRect.

    Args:
        contour (np.ndarray): Contour points.

    Returns:
        tuple: (x, y, w, h) - top-left corner (x, y), width (w), and
            height (h).
    """
    x_coords = [point[0] for point in contour]
    y_coords = [point[1] for point in contour]

    x_min, x_max = min(x_coords), max(x_coords)
    y_min, y_max = min(y_coords), max(y_coords)

    width = x_max - x_min
    height = y_max - y_min

    return x_min, y_min, width, height

def arc_length(contour, closed):
    """
    Custom implementation of cv2.arcLength.

    Args:
        contour (np.ndarray): Contour points.
        closed (bool): Whether the contour is closed.

    Returns:
        float: Perimeter of the contour.
    """
    length = 0.0
    for i in range(len(contour)):
        if i == len(contour) - 1 and not closed:
            break

        pt1 = np.array(contour[i])
        pt2 = np.array(contour[(i + 1) % len(contour)])
        length += np.linalg.norm(pt2 - pt1)

    return length
def normalize_contour(contour):
    """
    Normalizes contour format to (N, 2) array of float32.

    Args:
        contour: Input contour in any format (N, 1, 2) or (N, 2)
```

```
    Returns:
        np.ndarray: Normalized contour points as (N, 2) array
    """
    # Convert to numpy array if not already
    points = np.asarray(contour, dtype=np.float32)

    # Handle different contour formats
    if points.shape[-1] == 2: # Check if last dimension is 2 (x,y
        coordinates)
        if len(points.shape) == 3: # Format: (N, 1, 2)
            points = points.reshape(-1, 2)
        elif len(points.shape) == 2: # Format: (N, 2)
            pass
        else:
            raise ValueError(f"Unexpected␣contour␣shape:␣{points.shape}")
    else:
        raise ValueError(f"Contour␣must␣have␣2␣coordinates␣per␣point,␣got␣
            shape:␣{points.shape}")

    return points
def approx_poly_dp(contour, epsilon, closed):
    """
    Custom implementation of cv2.approxPolyDP.

    Args:
        contour (np.ndarray): Contour points.
        epsilon (float): Approximation accuracy.
        closed (bool): Whether the contour is closed.

    Returns:
        list: Approximated contour points.
    """
    def rdp(points, epsilon):
        """Ramer-Douglas-Peucker algorithm."""
        if len(points) < 3:
            return points

        start, end = np.array(points[0]), np.array(points[-1])
        max_dist = -1
        index = -1

        for i in range(1, len(points) - 1):
            pt = np.array(points[i])
            dist = np.abs(np.cross(end - start, pt - start) / np.linalg.
                norm(end - start))
            if dist > max_dist:
                max_dist = dist
```

```
            index = i

    if max_dist > epsilon:
        left = rdp(points[:index + 1], epsilon)
        right = rdp(points[index:], epsilon)
        return left[:-1] + right

    return [start.tolist(), end.tolist()]

def find_rectangle_groups(cnts):
    """
    Find groups of rectangles with similar x coordinates and check for count of 57
    """
    # First filter rectangles meeting basic criteria (x < 40 and w > h)
    rectangles_all=[]
    for cnt in cnts:
        rectangles_all.append(bounding_rect(cnt))
    filtered_rects = [rect for rect in rectangles_all if rect[0] < 40 and rect[2] > rect[3]]

    # Group rectangles by similar x coordinates
    x_groups = {}

    for rect in filtered_rects:
        x, y, w, h = rect
        # Check each x-1, x, and x+1
        found_group = False
        for base_x in range(x-1, x+2):
            if base_x in x_groups:
                x_groups[base_x].append(rect)
                found_group = True
                break

        if not found_group:
            x_groups[x] = [rect]

    # Print and check each group's count
    found_target = False
    print("\nChecking groups for count of 57:")
    for base_x, rects in sorted(x_groups.items()):
        count = len(rects)
        print(f"Group x={base_x}: {count} rectangles")

        if count == 57:
            print(f"Found group with exactly 57 rectangles at x={base_x}!")
            found_target = True
```

```
            # You might want to store or process this specific group
            target_group = rects
            return rects

    if not found_target:
        print("\nNo␣group␣with␣exactly␣57␣rectangles␣found.")
        # Print closest groups
        closest = min(x_groups.items(), key=lambda x: abs(len(x[1]) - 57))
        print(f"Closest␣group␣has␣{len(closest[1])}␣rectangles␣at␣x={closest[0]}")

    return x_groups

def find_centers_clustering(binary_image,rectangle,min_radius,max_radius):
    cropped_questions=crop_image(binary_image,rectangle[0],rectangle[1],rectangle[2],rectangle[3])

    cv2.imshow("crop␣ques",cropped_questions)
    cv2.waitKey(0)
    cropped_questions_not = cv2.bitwise_not(cropped_questions)

    cv2.imshow("crop␣ques␣not",cropped_questions_not)
    cv2.waitKey(0)
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
    cropped_questions_not=cv2.morphologyEx(cropped_questions_not,cv2.MORPH_OPEN,kernel)
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(8,8))
    cropped_questions_not=cv2.morphologyEx(cropped_questions_not,cv2.MORPH_OPEN,kernel)

    cv2.imshow("crop␣questions␣not␣procceseed", cropped_questions_not)
    cv2.waitKey(0)

    cropped_questions_processed=cv2.bitwise_not(cropped_questions_not)

    cv2.imshow("crop␣questions␣clustering␣input",cropped_questions_processed)
    cv2.waitKey(0)

    labels,centers=find_circle_centers(cropped_questions_processed,2,12)
    cluster_visulaize=visualize_labels_and_centers(cropped_questions_processed,labels,centers)
    cv2.imshow("␣clustering␣out", cluster_visulaize)
    cv2.waitKey(0)
    print(centers)
    print(len(centers))
    return centers
```

```
def map_answers_to_new_rectangle(original_corners, new_corners, answers):
    """
    Maps answers' pixel positions from the original rectangle to a new
       rectangle.

    Parameters:
        original_corners (list): List of 4 (x, y) tuples representing
           the corners of the original rectangle.
        new_corners (list): List of 4 (x, y) tuples representing the
           corners of the new rectangle.
        answers (list): List of (x, y) tuples representing the answers'
           pixel positions in the original rectangle.

    Returns:
        list: List of (x, y) tuples representing the answers' pixel
           positions in the new rectangle.
    """
    # Ensure inputs are numpy arrays
    original_corners = np.array(original_corners, dtype=np.float32)
    new_corners = np.array(new_corners, dtype=np.float32)

    # Compute the perspective transformation matrix
    transformation_matrix = cv2.getPerspectiveTransform(original_corners,
         new_corners)

    # Convert answers to homogeneous coordinates for transformation
    answers_array = np.array(answers, dtype=np.float32).reshape(-1, 1, 2)

    # Apply the perspective transformation
    transformed_answers_pixels = cv2.perspectiveTransform(answers_array,
       transformation_matrix)

    # Reshape the result back to a list of tuples
    return [tuple(point[0]) for point in transformed_answers_pixels]



def calculate_average_intensity(binary_image, gray_image,rectangles,
   mark_index,center,questions, radius=2):
    """
    Calculate the average intensity in a circular region around a center
       .

    Parameters:
        binary_image (numpy.ndarray): The binary image.
        center (tuple): The (x, y) center of the circular region.
```

```
    radius (int): The radius of the circular region.

Returns:
    float: The average intensity within the circular region.
"""

if questions:
    if mark_index%25==0:
        control_rectangle_index=0
    else:
        control_rectangle_index=25-((mark_index)%25)
    control_rectangle_x,control_rectangle_y,control_rectangle_w,
        control_rectangle_h=rectangles[control_rectangle_index]
    new_y=int(control_rectangle_y+control_rectangle_h/2)
    mask = np.zeros_like(binary_image, dtype=np.uint8)
    # gray_image=cv2.circle(gray_image,(int(center[0]), int(new_y))
        ,1,(255,255,0),1)
    # cv2.circle(mask, (int(center[0]), int(new_y)), radius, 255,
        -1)
    values = binary_image[mask == 255]
    # return float(np.mean(values)) if len(values) > 0 else 0
    x,y,w,h=int(center[0])-4,control_rectangle_y,8,control_rectangle_h
    roi = binary_image[y:y+h, x:x+w]
    gray_image[y:y+h, x:x+w]=[255,255,0]

# Calculate mean value
    mean_value = np.mean(roi)
    return mean_value

else:
    control_rectangle_x,control_rectangle_y,control_rectangle_w,
        control_rectangle_h=rectangles[mark_index]
    new_y=int(control_rectangle_y+control_rectangle_h/2)
    mask = np.zeros_like(binary_image, dtype=np.uint8)
    gray_image=cv2.circle(gray_image,(int(center[0]), int(new_y))
        ,1,(255,255,0),1)

    # cv2.circle(mask, (int(center[0]), int(new_y)), radius, 255,
        -1)
    values = binary_image[mask == 255]
    # return float(np.mean(values)) if len(values) > 0 else 0
    x,y,w,h=int(center[0])-4,control_rectangle_y,8,control_rectangle_h
    roi = binary_image[y:y+h, x:x+w]
    gray_image[y:y+h, x:x+w]=[255,255,0]
# Calculate mean value
    mean_value = np.mean(roi)
    return mean_value
```

```
def crop_image(img, x, y, width, height):
    # Read the image

    # Ensure coordinates are within image boundaries
    height_img, width_img = img.shape[:2]
    x = max(0, min(x, width_img))
    y = max(0, min(y, height_img))
    width = max(0, min(width, width_img - x))
    height = max(0, min(height, height_img - y))

    # Crop the image
    cropped = img[y:y+height, x:x+width]

    return cropped
```

F Main

```
    import numpy as np
import cv2
import pandas as pd
from contour import *
from canny import *
from cluster import *
from helpers import *
from adaptive_threshold import *

read_answers_df = pd.read_csv("/home/ibu/image_proces_report/codes/
    final_codes/true_answers.csv")
csv_path = '/home/ibu/image_proces_report/codes/final_codes/answers.
    pixels.csv' # Update with the correct path
uploaded_df = pd.read_csv("/home/ibu/image_proces_report/codes/
    final_codes/adjusted_circle_grid.csv", sep=';', index_col=0, encoding=
    'latin1')
read_answers_dict = dict(zip(read_answers_df['Answer␣Number'],
    read_answers_df['Answer']))
original_image=cv2.imread("/home/ibu/image_proces_report/codes/yeni_2.jpg
    ")
#

gtu_name_rectangle_ratio=((544-288)/(409-50))
gtu_question_rectangle_ratio=(543-253)/(726-423)
name_rectangle_index=0
question_rectangle_index=1
resized_image=None


blurred_image=None
```

```
resized_image=cv2.resize(original_image,(600,800), interpolation=cv2.
    INTER_AREA)


final_output=resized_image.copy()
gray_image=bgr_to_gray(resized_image)
gray_image_intensity_process=gray_image.copy()
gray_image_intensity_process[15:22, 12:17] = 255

for_output=resized_image.copy()



adaptive=adaptive_threshold(gray_image,255,adaptive_method="mean",
    threshold_type="binary",
        block_size=3, C=1)

adaptive2=adaptive_threshold(gray_image,255,adaptive_method="mean",
    threshold_type="binary",
        block_size=11, C=6)


adaptive_output_for_control=adaptive2.copy()


adaptive=medianBlur(adaptive2,1)

cv2.imshow("gray", gray_image)
cv2.waitKey(0)

cv2.imshow("adaptive", adaptive)
cv2.waitKey(0)

edges_reference_rectangles=apply_canny_edge_detection(False,
    adaptive_output_for_control,adaptive_output_for_control,
                          kernal_size=5,
                          low_threshold_ratio=0.05,
                          high_threshold_ratio=0.20,
                          gradient_estimation_filter_type="sobel")


edges=apply_canny_edge_detection(False,adaptive,adaptive,
                          kernal_size=5,
                          low_threshold_ratio=0.05,
                          high_threshold_ratio=0.20,
                          gradient_estimation_filter_type="sobel")
```

```
if edges.dtype == np.float16 or edges.dtype == np.int32:
    edges = cv2.normalize(edges, None, 0, 255, cv2.NORM_MINMAX)
    edges = np.uint8(edges)
cv2.imshow("edges from canny",edges)
cv2.waitKey(0)

contours, _ = findContours(edges)
contours = [np.array(contour, dtype=np.int32) for contour in contours]
con=gray_image.copy()
cv2.drawContours(con, contours, -1, (0, 255, 0), 1) # Yeil renkte (BGR
   formatnda) izin
cv2.imshow("contours ",con)
cv2.waitKey(0)

contours_reference_rectangles, _ = findContours(
   edges_reference_rectangles)
contours_reference_rectangles = [np.array(contour, dtype=np.int32) for
   contour in contours_reference_rectangles]



rectangles_reference=[]
gray_image=cv2.cvtColor(gray_image, cv2.COLOR_GRAY2BGR)

rect_reference=gray_image.copy()

rect_reference2=gray_image.copy()
groups = find_rectangle_groups(contours_reference_rectangles)

if isinstance(groups,list) and len(groups)==57:
    rectangles_reference=groups
elif isinstance(groups,dict):
    rectangles_reference = min(groups.items(), key=lambda x: abs(len(x
        [1]) - 57))

rectangles_reference.sort(key=lambda rect: rect[1],reverse=True)

for rctn in rectangles_reference:
        x, y, w, h = rctn

        cv2.rectangle(gray_image, (x, y), (x + w, y + h), (255, 255, 0),
            1)
        gray_image=cv2.line(gray_image,(0,y),(gray_image.shape[1],y)
            ,(255,0,0),1)
        gray_image=cv2.line(gray_image,(0,y+h),(gray_image.shape[1],y+h)
```

```
            ,(255,0,0),1)
rectangles_all=[]
min_area = 500 # Define the minimum area threshold
largest_area=0
question_rectangle=None

for contour in contours:
    # Get the bounding rectangle for the contour
    x, y, w, h = bounding_rect(contour)
    area = w * h

    # Draw the rectangle if the area is larger than the threshold
    if area > min_area:
        epsilon = 0.02 * arc_length(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)

    # Check if it has 4 vertices and is convex
        if len(approx) == 4 :
            cv2.rectangle(gray_image, (x, y), (x + w, y + h), (0, 255, 0),
                 1) # Green rectangle
            print(x,y,w,h)
            rectangles_all.append((x,y,w,h))

cv2.imshow("gray_rectangeles2", gray_image)
cv2.waitKey(0)

max_sum = -float("inf")

max_sum_name = -float("inf")

question_rectangle=None
name_rectangle=None

rectangles_all = sorted(rectangles_all, key=lambda rect: rect[2] * rect
    [3],reverse=True)
name_rectangle=rectangles_all[name_rectangle_index]
question_rectangle=rectangles_all[question_rectangle_index]
if gtu_name_rectangle_ratio*0.98<(name_rectangle[2]/name_rectangle[3])<
   gtu_name_rectangle_ratio*1.02:
    if not abs(name_rectangle[2] - question_rectangle[2])<=3:
        # question_rectangle[2]=name_rectangle[2]+name_rectangle[0]-
            question_rectangle[0]
        question_rectangle=(question_rectangle[0],question_rectangle[1],
           name_rectangle[2]+name_rectangle[0]-question_rectangle[0],
           question_rectangle[3])
elif gtu_question_rectangle_ratio*0.98<(question_rectangle[2]/
   question_rectangle[3])<gtu_question_rectangle_ratio*1.02:
        name_rectangle=(name_rectangle[0],name_rectangle[1],
```

```
            question_rectangle[2]+question_rectangle[0]-name_rectangle[0],
            name_rectangle[3])




answers_df = pd.read_csv(csv_path)
answer_sheet_pixels = list(zip(answers_df['X'], answers_df['Y']))

questions = answers_df['Question'].tolist()
options = answers_df['Option'].tolist()

question_rectangle_original_corners = [(1, 0), (730, 1), (732, 822), (2,
   822)]

 # Define the new rectangle corners (top-left, top-right, bottom-right,
    bottom-left)
x,y,w,h=question_rectangle
question_rectangle_new_corners = [(x,y), (x+w, y), (x+w, y+h),(x, y+h)]
#print(new_corners)



rectangles_reference.sort(key=lambda rect: rect[1],reverse=True)

question_contol_rectangles=rectangles_reference[:25]
centers=find_centers_clustering(adaptive2,question_rectangle,2,12)

centers_global=[]
for center in centers:
    centers_global.append((center[0]+question_rectangle[0],center[1]+
       question_rectangle[1]))

result = {}

    # Initialize result dict with rectangle y values
for rect in question_contol_rectangles:
    y = rect[1] # y coordinate of rectangle
    result[y] = []

# For each y, find closest 4 centers
for y in result.keys():
    # Calculate distances from this y to all centers
    distances = [(center, abs(center[1] - y)) for center in
       centers_global if abs(center[1] - y) < 5]
    # Sort by distance and take closest 4
    closest = sorted(distances, key=lambda x: x[1])[:4]
    result[y] = [center for center, dist in closest]
```

```
i=0
for rectangle in rectangles_reference:
    x,y,w,h=rectangle
    rect_reference2=cv2.line(rect_reference2,(0,y),(rect_reference2.shape
        [1],y),(255,0,0),1)
    rect_reference2=cv2.line(rect_reference2,(0,y+h),(rect_reference2.
        shape[1],y+h),(255,0,0),1)

cv2.imshow("before␣rect␣process",rect_reference2)
cv2.waitKey(0)
for y in result.keys():
    original_rect=rectangles_reference[i]
    centers=result[rectangles_reference[i][1]]
    first_elements = [t[1] for t in centers]
    new_y=sum(first_elements) / len(first_elements)
    rectangles_reference[i]=(original_rect[0],int(new_y-(original_rect
        [3]/2)),original_rect[2],original_rect[3])
    i+=1
for rectangle in rectangles_reference:
    x,y,w,h=rectangle
    rect_reference=cv2.line(rect_reference,(0,y),(rect_reference.shape
        [1],y),(255,0,0),1)
    rect_reference=cv2.line(rect_reference,(0,y+h),(rect_reference.shape
        [1],y+h),(255,0,0),1)

cv2.imshow("after␣rect␣process",rect_reference)
cv2.waitKey(0)

transformed_answers_pixels = map_answers_to_new_rectangle(
    question_rectangle_original_corners, question_rectangle_new_corners,
    answer_sheet_pixels)
# print(transformed_answers_pixels)
# for point in transformed_answers_pixels:
#     gray_image=cv2.circle(gray_image,(int(point[0]),int(point[1]))
    ,1,(255,255,0),1)
    # Combine data into a dictionary structure
question_dict = {}
correct_answer=0
for i in range(len(answer_sheet_pixels)):
    question_num = questions[i]
    option = options[i]
    # print(question_num,i)
    if question_num not in question_dict:
        question_dict[question_num] = {}
    question_dict[question_num][option] = {
        "Original": answer_sheet_pixels[i],
        "Transformed": transformed_answers_pixels[i],
       "Intensity": calculate_average_intensity(adaptive2,final_output,
```

```
            rectangles_reference,question_num,transformed_answers_pixels[i
            ],True)

    }
    if option=='E':
        min_intensity=min(question_dict[question_num].items(), key=lambda
            item: item[1]["Intensity"])
        #print(min_intensity)
        question_dict[question_num]["Marked"]=min_intensity[0]
        true_option = read_answers_dict.get(question_num-1)
        is_correct = question_dict[question_num]['Marked'] == true_option
        question_dict[question_num]["is_correct"]=is_correct

        if is_correct:
            correct_answer+=1

        #print(is_correct,question_num,question_dict[question_num]["
            Marked"],true_option)
#print(correct_answer)
# print(question_dict)
for question_num in sorted(question_dict.keys()):
    marked = question_dict[question_num]["Marked"]
    is_correct = question_dict[question_num]["is_correct"]

    print(f"{question_num:<10}␣{marked:<10}␣{''␣if␣is_correct␣else␣''}")
print(f"True␣Marks:␣{correct_answer}␣False␣Marks:␣{100-correct_answer}")
for question_num in sorted(question_dict.keys()):
    marked = question_dict[question_num]
    # print(f"{question_num} {marked}")
# for entry in transformed_answers_pixels:
#         x, y = int(entry[0]), int(entry[1])
#         cv2.circle(resized_image, (x, y), 3, (255, 255, 0), -1)

def transform_point_to_new_rectangle(original_corners, new_corners, point
    ):
    """
    Maps a single point's pixel position from the original rectangle to
        a new rectangle.

    """
    import numpy as np
    import cv2

    # Ensure inputs are numpy arrays
    original_corners = np.array(original_corners, dtype=np.float32)
    new_corners = np.array(new_corners, dtype=np.float32)
    point_array = np.array([[point]], dtype=np.float32)
```

```
    # Compute the transformation matrix
    transformation_matrix = cv2.getPerspectiveTransform(original_corners,
        new_corners)

    # Apply the perspective transformation
    transformed_point = cv2.perspectiveTransform(point_array,
       transformation_matrix)

    # Extract and return the transformed point as a tuple
    return tuple(transformed_point[0][0])
original_name_corners=[(26,33),(896,40),(900,1308),(30,1305)]




x_n,y_n,w_n,h_n=name_rectangle
name_rect_image=[(x_n,y_n),(x_n+w_n,y_n),(x_n+w_n,y_n+h_n),(x_n,y_n+h_n)]
cv2.rectangle(gray_image,(x_n,y_n),(x_n+w_n,y_n+h_n),(255,255,0),1)
# Initialize the dictionary to store the data
column_dictionary = {}
turkish_alphabet = ['A', 'B', 'C', '', 'D', 'E', 'F', 'G', '', 'H', 'I',
   '', 'J', 'K', 'L', 'M', 'N', 'O', '', 'P', 'R', 'S', '', 'T', 'U', '',
    'V', 'Y', 'Z']

# Process each column and row
# aa=cv2.imread("images/ex_13.png")
# cv2.imshow("aaaa",aa)
# cv2.waitKey(0)
chars=[]
columns={}
a=0
rectangles_reference.sort(key=lambda rect: rect[1],reverse=False)
for column in uploaded_df.columns:
    column_data = {} # To store real and transformed pixel values for
       each entry in the column
    alphabet_index = 0 # Reset alphabet index for each column

    for row in uploaded_df.index:
        cell = uploaded_df.at[row, column]
        if pd.notna(cell): # If the cell is not empty
            # Parse the coordinates from the string
            coords = cell.strip("()").split(", ")
            x, y = int(coords[0]), int(coords[1])

            chars.append((x,y))
           # print(x,y)
            # print(alphabet_index,a)
            (x1,y1)=transform_point_to_new_rectangle(original_name_corners
```

```
            ,name_rect_image,(x,y))
        column_data[str(turkish_alphabet[alphabet_index% len(
            turkish_alphabet)])]={}
        column_data[str(turkish_alphabet[alphabet_index% len(
            turkish_alphabet)])]={

            "real_pixel": (x, y),
            "transformed_pixel": (x1,y1),
            "Intensity" : calculate_average_intensity(adaptive2,
                final_output,rectangles_reference,alphabet_index,(x1,y1
                ),False,2)
        }
        # cv2.circle(gray_image, (int(x1), int(y1)), 3, 255, -1)

        #print(column_data[str(turkish_alphabet[alphabet_index% len(
            turkish_alphabet)])])
        alphabet_index+=1
       # print(len(column_data ))

    columns[a]=column_data

    a+=1

# print(columns)

name=[]
for key in columns.keys():
    min_intensity=min(columns[key].items(), key=lambda item: item[1]["
        Intensity"])
    # print(min_intensity)
    data=min_intensity[1]
    if min_intensity[1]['Intensity']<75:
        columns[key]["Min␣Intensity"]=min_intensity[0]
        name.append(min_intensity[0])
    else:
        columns[key]["Min␣Intensity"]="␣"
        name.append("␣")
    # column_dictionary[column] = column_data
    # print(column_data)


cv2.imshow("final_output", final_output)
# for question_num in sorted(question_dict.keys()):
#     marked = question_dict[question_num]["Marked"]
#     is_correct = question_dict[question_num]["is_correct"]

#     print(f"{question_num:<10} {marked:<10} {'' if is_correct else '
    '}")
```

```
cv2.waitKey(0)
print(name)
rectangles_reference.sort(key=lambda rect: rect[1])
```