Digit recognition from image

CSE-4128: Image Processing and Computer Vision Laboratory

Submitted to:

Dr. Sk. Md. Masudul Ahsan
Professor,
Department of CSE, KUET

Dipannita Biswas
Lecturer,
Department of CSE, KUET

Submitted by:

Md Abu Saeed
Roll: **1907057**
Dept. of CSE, KUET

1 Objectives

- To develop a system that is capable of recognizing digits(0-9).
- To learn and apply different image processing techniques.

2 Introduction

The primary focus of this project is to extract and identify distinct segments within an image(digits(0-9)) and apply different image processing techniques to analyze these segments and produce results.

The project begins by allowing users to input an image through a dropdown menu. Once an image is selected, it is read into the system using the OpenCV library, a tool for image processing operations. The first step in the analysis is converting the image into grayscale. This simplification reduces complexity by eliminating color variables, focusing instead on intensity and contrast within the image.

A key aspect of the project's approach is its assumption that the background of the image has a constant intensity and is the most common intensity throughout the image. By identifying and counting this dominant intensity, it can effectively isolate the background by converting these areas to white (intensity value of 255).

Once the background is isolated, the algorithm searches for black pixels, which indicate the presence of foreground elements. Starting from these black pixels, it uses a Depth-First Search (DFS) algorithm to go through connected pixels, identifying the boundaries of each distinct segment. This process is crucial for accurately determining the shapes and sizes of the elements within the image.

After identifying these segments, it draws rectangles around each one, visually marking them for further analysis. It then extracts each segment individually, taking care to separate any overlapping elements using further DFS operations. This segmentation ensures that each segment is analyzed independently.

The extracted segments undergo a series of rotations, from 0 to 360 degrees at 2-degree intervals. For each rotation, the algorithm calculates the area of the bounding rectangle, prioritizing configurations that minimize the width and then the area, which are crucial for aligning the segment.

Then template matching is used which involves comparing best three rotated segment against a set of predefined templates to find the best match. This comparison is vital for identifying and annotating each segment within the original image accurately. Finally, the software annotates the matched segments directly on the image, drawing empty rectangles around them to highlight the identified elements.

Finally a generated image is shown which contains all the identifing digits on that images.

3 Tools used

The following tools are used in the project:

- Programming language
  - Python
- Libraries
  - **cv2** for image read, write, resize
  - **numpy** for fast and easier calculation in array.
  - **colorsys** for generating random color for annotating.
  - **math** for few mathematical operations and constant.
  - **collections** for datastructure like defaultdict, deque etc.
- Interface
  - **tkinter** for user interface.
  - **PIL** for making image suitable for tkinter.

4 Methodology

The project works as follows:

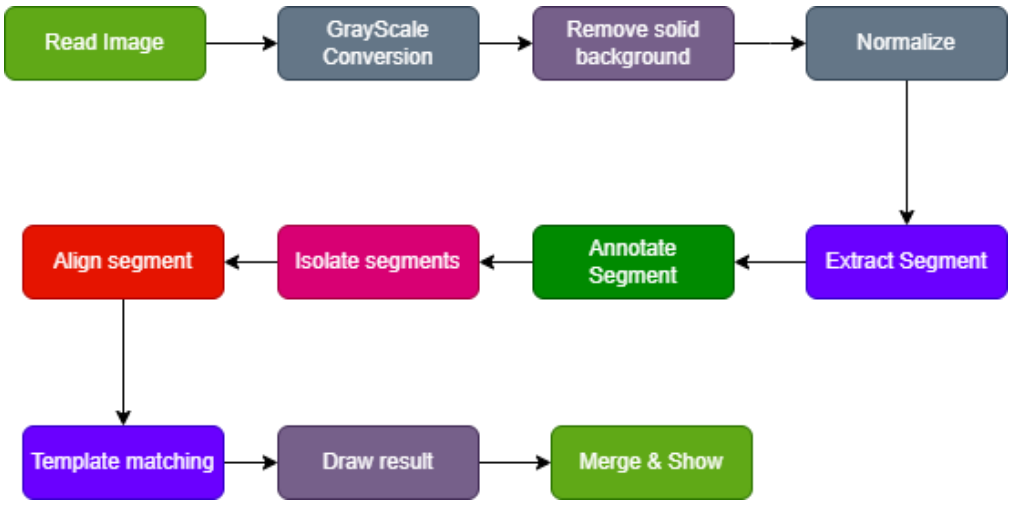

Figure 1: Flow chart

4.1 Read Image & gray-scale conversion

1. User select an image from a predefined images and the **path** is returned by tkinter.
2. Then image is read and converted using **cv2** library.

```
image = cv2.imread(path)
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

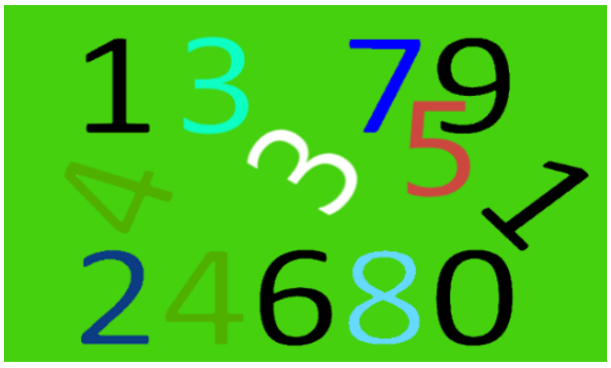

Figure 2: Input image

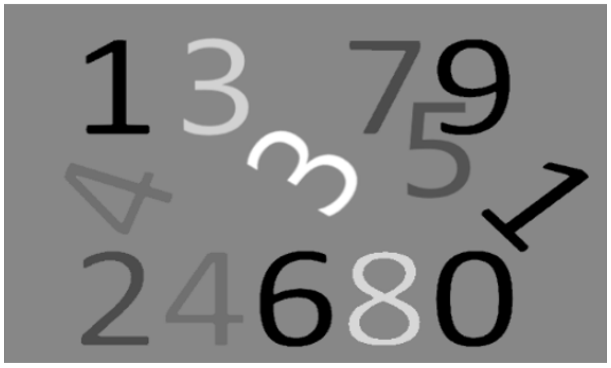

Figure 3: Gray-scale converted

4.2 Remove solid background

1. It count the intensity that appears most.
2. Then it replace the old intensity with White(255).
3. It also change other intensities to Gray(50) to make them highlighted.

Pseudo-code:

```
count_map = defaultdict(int)
h,w = image.shape

count intensity and store it in count_map

find the maximum intensity from count_map

copy the image to new_image

iterate all pixel(x,y) and do the following
    if pixel_val == inten:
        new_image[x,y] = WHITE
    else:
        new_image[x,y] = GRAY
return new_image
```

Result:

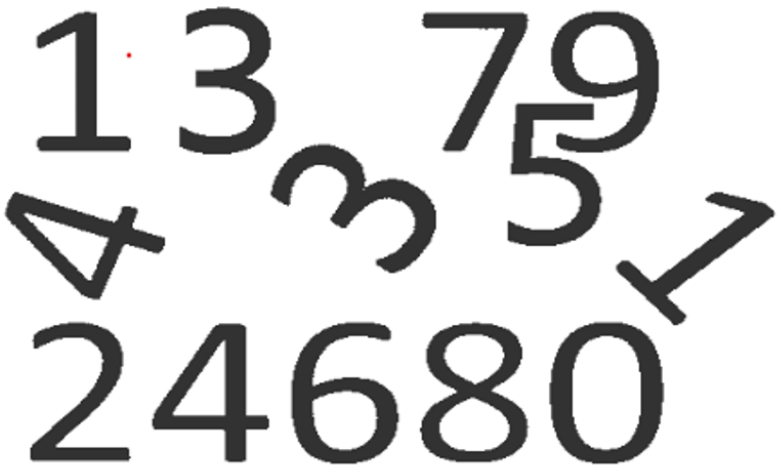

Figure 4: Back ground removed and highlighted

4.3 Extract segment

1. Iterate the whole image and find black pixel.
2. Apply DFS on that black pixel and extract four corner points for that digit.
3. Continue until all black pixels are visited.

Class for holding one segment:

```
class MySegment:
    def __init__(self, rect, blackPoint):
        self.rect = rect
        self.blackPoint = blackPoint
```

Pseudo-Code for extracting segment:

```
def segment_new(image):

    Make a copy of the image
    Initialize a empty list for 'segments'

    Iterate all the pixels(x,y) and do the following
        if image[x,y] == 0: # black
            min_x, min_y, max_x, max_y = spread(image,x,y)

            Create four corner points p1, p2, p3, p4

            rect = (p1,p2,p3,p4)

            segment = MySegment(rect=rect, blackPoint=(x,y))
            segments.append(segment)
    return image, segments
```

Pseudo-Code for spread function:

```
def spread(image,x,y):
    Initialize min_x, max_x, min_y, max_y with x,x,y,y
    Initialize queue and add (x,y)
    Initialize black_points and add (x,y)

    area = []
    while queue is not empty:
        x,y = queue.popleft()

        if image[x,y] != 0:
            skip this itearation and continue

        min_x, max_x = min( min_x, x ), max( max_x, x )
        min_y, max_y = min( min_y, y ), max( max_y, y )

        Add (x,y) to area and update image[x,y] = 120

        for each adj valid pixel(nx,ny) of (x,y), do these:
            if image[nx,ny] == 0:
                queue.append( (nx,ny) )
                black_points.append( (nx,ny) )
    return min_x, min_y, max_x, max_y
```

Result:

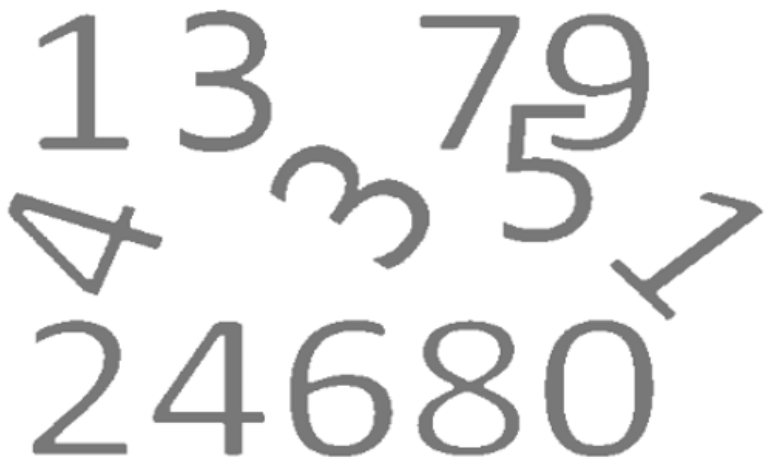

Figure 5: Extracted segments

4.4 Annotate segment

1. Iterate all the segments one by one.
2. Draw the bounding box around the segment.
3. Continue until all black pixels are visited.

Code for annotating:

```
def annotate_rect_points(image, my_segments):
    image = image.copy()

    for seg in my_segments:
        draw_rect_at(image, seg.rect[0], seg.rect[3],
                             seg.rect[2], seg.rect[1],80)
    return image
```

Code for drawing single boundary box:

```
def draw_rect_at(image, top_left, top_right, bottom_right, bottom_left,
↪   INTEN):

    # Line from top_left to top_right
    for y in range(top_left[1], top_right[1]):
        image[ top_left[0], y ] = INTEN

    # Line from top_right to bottom_right
    for x in range(top_left[0], bottom_left[0]):
        image[ x, top_left[1]] = INTEN

    # Line from bottom_right to bottom_left
    for x in range(top_right[0], bottom_right[0]):
        image[ x, top_right[1]] = INTEN

    # Line from bottom_left to top_left
    for y in range(bottom_left[1], bottom_right[1]):
        image[ bottom_left[0], y ] = INTEN
```

Result:

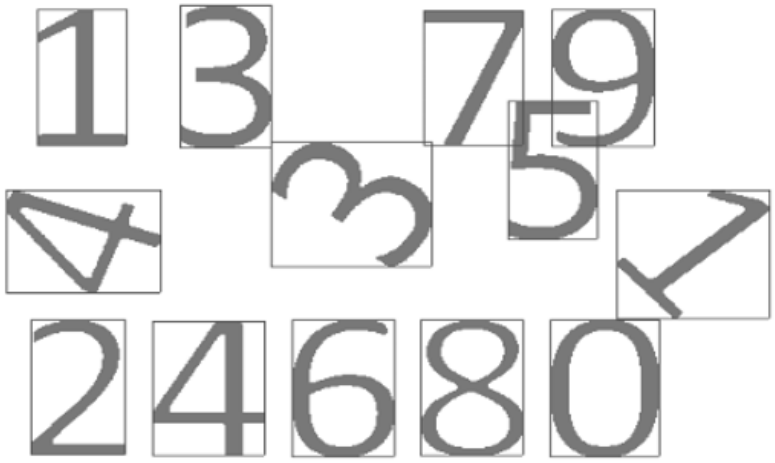

Figure 6: Annotated segments

4.5 Isolate segment

1. Extract a segment area for further processing.

Pesudo-Code for extracting one segment:

```
# `seg` area will be extracted from `image`
def get_image_at_segment(image, seg):
    top_left = seg.rect[0]
    bottom_right = seg.rect[2]

    # Height and widht of the segment
    h = bottom_right[0] - top_left[0] + 2
    w = bottom_right[1] - top_left[1] + 2

    Initialize empty segment of 255 of size (h,w)

    h,w = image.shape

    x = seg.blackPoint[0]
    y = seg.blackPoint[1]

    Initialize min_x, max_x, min_y, max_y with x,x,y,y
    Initialize queue and add (x,y)

    # Initialize starting position
    sx,sy = x - top_left[0], y - top_left[1]
    segment[sx][sy] = 0

    area = []
    while queue is not empty:
        x,y = queue.popleft()
        if image[x,y] != 0:
            skip and continue to next iteration

        min_x, max_x = min( min_x, x ), max( max_x, x )
        min_y, max_y = min( min_y, y ), max( max_y, y )

        Add (x,y) to area and mark image[x,y] visited

        for each adj valid pixel(nx,ny) of (x,y), do these:
            if image[nx,ny] == 0:
                Add (nx,ny) to queue

                sx, sy = nx - top_left[0], ny - top_left[1]
                segment[sx][sy] = 0 # copied
    return segment
```

Result:

Figure 7: Two isolated segments from previous image

4.6 Align segment

1. Rotate the segment from (0-360) at interval 2.
2. Compute the area for each rotation.
3. Calculate the first two segments with minimum width then area.
4. The minimum result will be used in template matching.

Pesudo-Code for aligning one segment:

```
def get_aligned_digit(segment):
    points = extract_points(segment)
    Init temp_points, best_points, best_points_two to points
    w, h, lx, ly = get_area_param(points)

    min_area, best_angle = w * h, 0
    rotated_states = []

    for angle in range(0,360, 2):
        points = rotate_points(temp_points, angle)
        nw, nh, nlx, nly = get_area_param(points)

        area = nw * nh

        if abs(min_area-area) <= 500:
            rotated_states.append( RotatedState( nw, nh, points, nlx, nly,
             ↪  area, angle ) )

            min_area, best_angle = area, angle
            best_points = points
            w,w,lx,ly = nw,nh,nlx,nly

    rotated_states = sorted(
                rotated_states,
                key=lambda state: (state.width,state.height)
        )

    image_one = None
    image_two = None

    state = rotated_states[0]
    image_one = plot_points_on_image(
            state.height, state.width,
            state.points, state.left_x, state.top_y
        )
    image_one = fill_whole(image_one)
```

```
    state = rotated_states[1]
    image_two = plot_points_on_image(
        state.height, state.width,
        state.points, state.left_x, state.top_y
    )
    image_two = fill_whole(image_two)

    return image_one, image_two
```

extract-points function:

```
def extract_points(segment):
    points = []

    h,w = segment.shape

    for x in range(h):
        for y in range(w):
            if(segment[x,y] == BLACK):
                points.append( (x,y) )
    return points
```

get-area-param function:

```
def get_area_param(points):
    min_x, min_y, max_x, max_y = inf, inf, -inf, -inf

    for pt in points:
        min_x, min_y = min(min_x, pt[0]), min(min_y, pt[1])
        max_x, max_y = max(max_x, pt[0]), max(max_y, pt[1])

    w,h = max_y - min_y, max_x - min_x
    return w, h, min_x, min_y # Area
```

rotate-points function:

```
def rotate_points(points, angle):
    angle_rad = math.radians(angle)
    ox, oy = (0,0)
    rotated_points = []

    cos_theta, sin_theta =
            math.cos(angle_rad), math.sin(angle_rad)

    for x,y in points:
        tx, ty = x - ox, y - oy
        # Apply rotation
        rotated_x = int(math.ceil(
                    tx * cos_theta - ty * sin_theta
                )
            )
        rotated_y = int(math.ceil(
                    tx * sin_theta + ty * cos_theta
                )
            )

        # Reverse translation
        final_x,final_y = rotated_x + ox, rotated_y + oy
        rotated_points.append((final_x, final_y))

    return rotated_points
```

RotatedState class:

```
class RotatedState:
    def __init__(self,width, height, points,
                    left_x, top_y, area, angle):
        self.width = width
        self.height = height
        self.points = points
        self.left_x = left_x
        self.top_y = top_y
        self.area = area
        self.angle = angle
```

plot-points-on-image function:

```
def plot_points_on_image(h,w, best_points, left_x, top_y):
    image = 255 * np.ones((h+1,w+1), dtype=np.uint8)

    for pt in best_points:
        x = pt[0] - left_x
        y = pt[1] - top_y

        image[x][y] = BLACK
    return image
```

fill-whole function:

```
def fill_whole(old_image):
    h,w = old_image.shape
    image = old_image.copy()

    Iterate all pixels (x,y) and do these:
        if old_image[x,y] != WHITE:
            skip iteration

        Iterate the valid adjacent pixel (nx,ny), do these:
            if old_image[nx,ny] == BLACK:
                image[x,y] = BLACK
                break
    return image
```

Result:

Figure 8: Original(left) and two segment with minimum area

4.7 Template matching

1. Some predefined images are used template.
2. Compares the actual and minimum segments with templates and calculate best match.

Pesudo-Code for template matching:

```
def perform_matching(segment, use_actual = False):
    segment = segment.copy()
    templates = []

    h,w = segment.shape
    if segment size is larger than template size, then
        Resize the segment to template size
        templates = read_templates(use_actual=use_actual)
        h,w = TEMPLATE_SIZE[1], TEMPLATE_SIZE[0]
    else:
        h,w = min(TEMPLATE_SIZE[1],h), min(TEMPLATE_SIZE[0],w)
        segment = cv2.resize(segment,(w,h),segment)
        templates = read_templates(h=h, w=w, use_actual=use_actual)

    best_template, best_segment = (-1,0), None
    percents = []
    for each template in templates, do these:
        Calc number of matched pixel and store in `matched` var
        Add percent to percents list
        Update best_template, best_segment if this is better.

    percent = ( 100 * best_template[1] ) / (h * w)
    return best_template[0], best_segment, percent, percents
```

Code for reading template:

```
def read_templates(h=-1,w=-1, use_actual=False):
    templates = []
    for x in range(10):
        path = "min_area_template\\"+str(x)+".png"
        if use_actual:
            path = "actual_template\\"+str(x)+".png"

        image = cv2.imread(path,0)
        if h != -1 and w != -1:
            image = cv2.resize(image,(w,h),image)
        templates.append(image)
    return templates
```

Result:

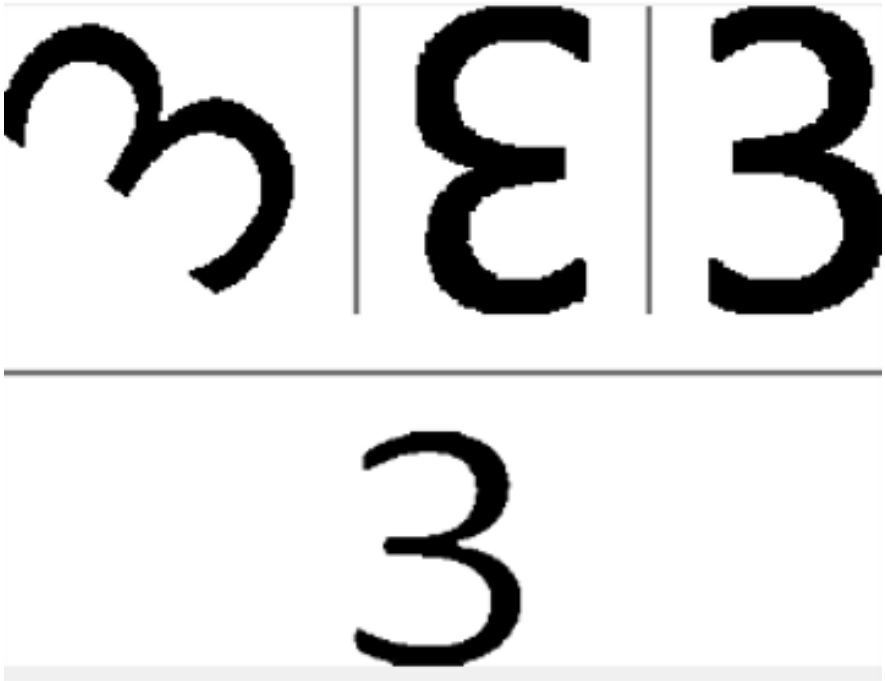

Figure 9: Segments(top) and matched template(bottom)

4.8 Final Result

1. Template matching is performed for each segments.
2. An empty rectangle is drawn at corresponding position.
3. Each rectangle contains the detected digit with color.

Pseudo-Code for final result creation:

```
def extract_and_detect_segments(gray_image, my_segments):
    Initialize an empty color image same size as gray_image

    index = 0
    for each seg in my_segments:
        rect = Take the bounding rectangle in `seg`
        segment = Extract the segment at `rect` in gray_image

        digit_one, digit_two = get_aligned_digit(segment)

        matched_dig_act, matched_seg_act,
            percent_act, pcnts = perform_matching(
                        segment=segment, use_actual=True
                    )
        matched_dig_one, matched_seg_one,
            percent_one, _ = perform_matching(
                        segment=digit_one
                    )

        matched_dig_two, matched_seg_two,
            percent_two, _ = perform_matching(
                        segment=digit_two
                    )

        matched_dig, matched_seg,
                    percent = Take the best among them

        actual_template = get_actual_template(matched_dig)

        merged_image = merge_images(
                image1=segment, image2=digit_one, horiz=True)

        merged_image = merge_images(
         ↪  image1=merged_image,image2=digit_two,horiz=True)

        merged_image = merge_images(
                merged_image, actual_template, horiz = False)
```

```
color = generate a random color for this position

Draw rectangle at the colored image
Draw rectangle at the empty image
Draw text at the empty image
```

get-actual-template function:

```
def get_actual_template(digit):
    path = "actual_template\\"+str(digit)+".png"
    image = cv2.imread(path,0)
    return image
```

Pseudo-Code for merging function:

```
def merge_horiz(image1, image2):
    new_height = max height of two
    new_width = sum of widths

    merged_image = Empty white image with new size

    pad = (new_height - height of image1) // 2

    # Copy over the first image
    for each pixel(x,y)
        merged_image[x+pad][y] = image1[x][y]

    # vertical gap of width 4px
    for x in range(new_height):
        for i in range(10, 30):
            merged_image[x][width1+i] = 120

    # Copy over the second image
    pad = (new_height - height of image2) // 2
    for each pixel(x,y):
        merged_image[x+pad][width1 + 40 +y] = image2[x][y]

    return merged_image

def merge_vert(image1, image2):
    new_width = max widths of two images
    new_height = sum of heights

    merged_image = Empty white image with new size

    pad = (new_width - width of image1) // 2

    # Copy over the first image
    for each pixel(x,y):
        merged_image[x][y+pad] = image1[x][y]

    # horiz gap of width 4px
```

```
    for y in range(new_width):
        for i in range(10, 30):
            merged_image[height1+i][y] = 120

    # Copy over the second image
    pad = ( new_width - width of image2) // 2
    for each pixel(x,y):
        merged_image[height1+40+x][y+pad] = image2[x][y]

    return merged_image


def merge_images(image1, image2, horiz = False):
    if horiz:
        return merge_horiz(image1,image2)
    else:
        return merge_vert(image1, image2)
```

Result:

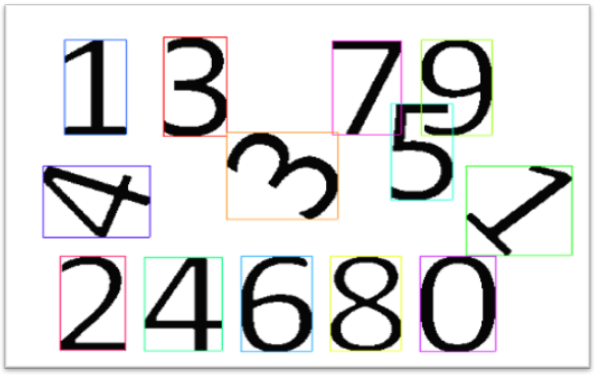

Figure 10: Annotated input

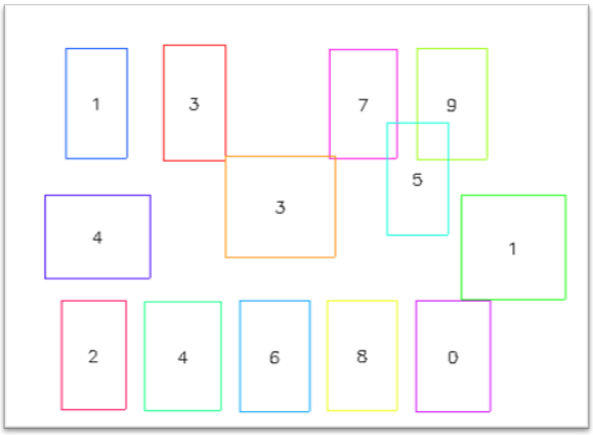

Figure 11: Final result

4.9 GUI

1. Some predefined images are used template.
2. Compares the actual and minimum segments with templates and calculate best match.
3. Select from predefined image.
4. Step-wise preview and output.
5. Next and Prev button to see next and previous step results.

Result:

Figure 12: Graphical User Interface)

5 Discussion

In this image processing project, I utilized various image processing techniques for digit. By converting the images to gray-scale and isolating the foreground, I simplified the initial stages of the image analysis. This setup is needed as it ensures clarity and make the image simple for processing.

I used Depth-First Search (DFS) that allows for precise segmentation by accurately mapping and extracting each element, ensuring that every segment is processed correctly without overlap from adjacent elements. However, my assumption that the background intensity is always constant could pose limitations, particularly in scenarios where the background varies can lead to recognition errors.

To enhance the accuracy of the project, I implemented a process where each segmented part is rotated and analyzed to determine the optimal orientation for template matching. This technique is vital for matching each segment against predefined templates accurately.

6 Conclusion

In conclusion, this image processing project effectively demonstrates the application of image processing techniques tailored for recognizing digits in various images. By utilizing methods such as gray-scale conversion, which simplifies the visual data, and employing Depth-First Search (DFS) to ensure accurate segmentation.

The addition of rotational analysis to determine the optimal orientation for template matching further enhances the capability to accurately match and identify each digit.

The project has some limitation as well. It requires a solid background and cannot handle multiple fonts or overlapping digits, which may lead to incorrect results. This restricts its effectiveness in more complex visual scenarios.