Sudoku Solver

The objective of this project is to solve a Sudoku puzzle given a picture from https://www.sudoku-online.org/.

To achieve this, we divide the process into three steps.

Firstly, we extract individual images of each square.

Next, we identify the number within each square to construct a vector that represents the Sudoku puzzle.

Lastly, we employ a backtracking algorithm to solve the Sudoku puzzle using this vector.

Frist step: Individual Images

You can find a comprehensive guide on Canny edge detection at this link.

Initially, we employ Canny edge detection to recognize the borders of the Sudoku board.

The algorithm for Canny edge detection typically involves five steps:

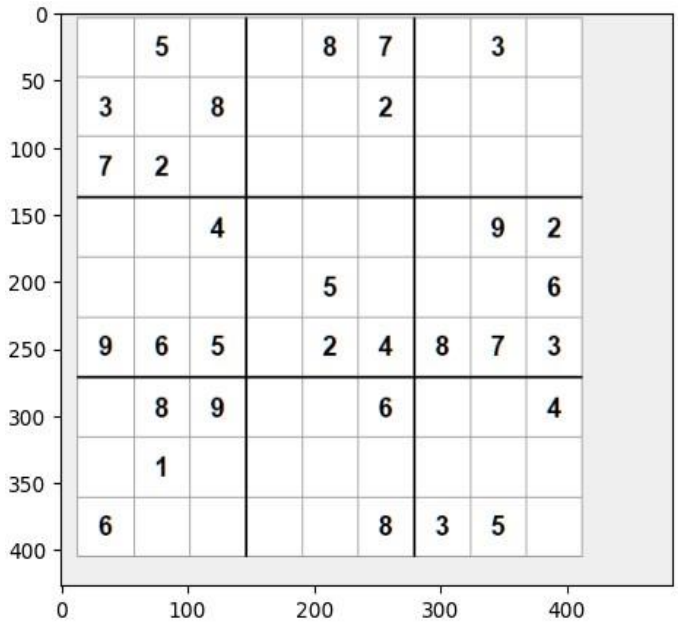
Noise Reduction: Eliminate outliers to facilitate gradient calculations.

Gradient Calculation: Utilize Sobel filters to discern changes in pixel intensity.

Non-maximum Suppression: Adjust the value of all edges based on neighboring pixels.

Double Threshold: Implement dual thresholds to differentiate between strong and weak pixels.

Edge Tracking by Hysteresis: Elevate the strength of weak pixels in proximity to strong ones.



Original Picture

```
def DetectEdge(path,thr1,thr2):
    "Apllies CannyEdgeDetection and returns original image and coc
    img = cv.imread(path)
    img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    #Use Canny Edge Detection Algorithm
    edged = cv.Canny(img_gray, thr1, thr2)
    #get contours from the edged image
    contours, _ = cv.findContours(edged, cv.RETR_EXTERNAL,
                                  cv.CHAIN_APPROX_SIMPLE)
    return img,contours
```

The OpenCV function simplifies the process by applying the algorithm in a single step. We have the flexibility to specify the thresholds for the fourth step, known as the double thresholding phase.

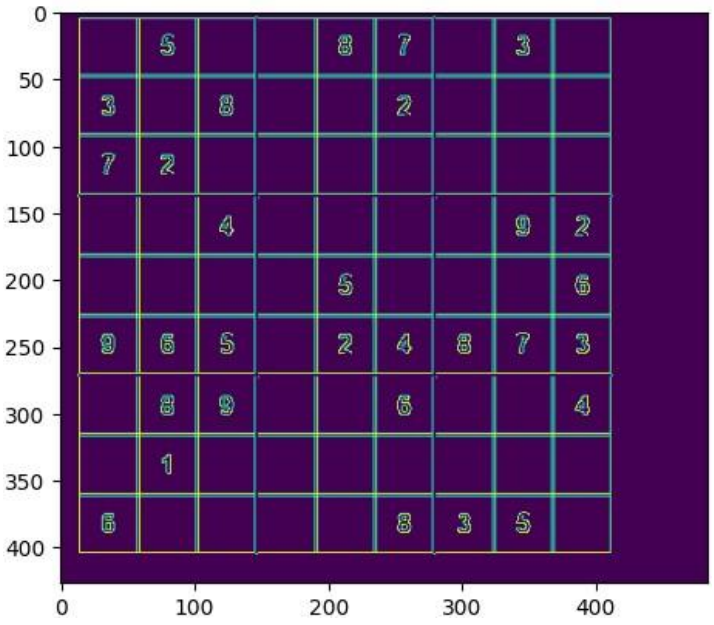


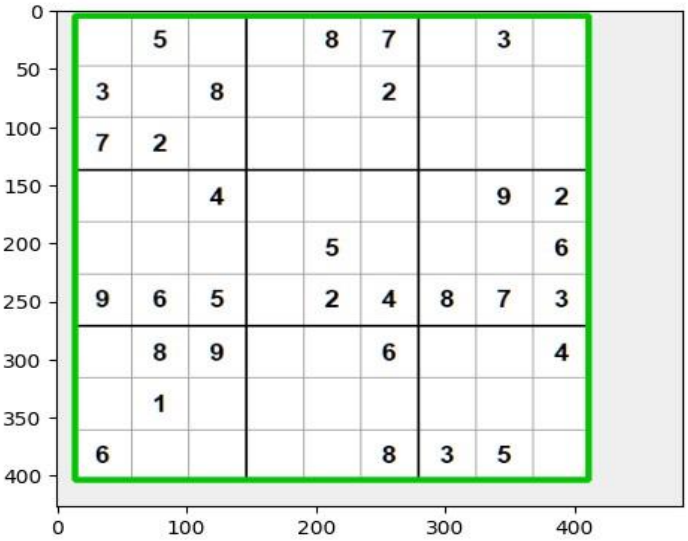
Image after canny edge detection algorithm

Next, we identify the contours of the board. OpenCV provides a function specifically for this purpose. This function detects changes in color within an image. When numerous points align to form a continuous line with consistent color and intensity, we recognize it as a contour.

```
w, h = img.shape[1], img.shape[0]
for cntr in contours:
    imgx, imgy, imgw, imgh = cv.boundingRect(cntr)
    if imgw < w/5 or imgw < h/5 or imgw/imgh < 0.25 or imgw/imgh > 1.5:
        continue
  # Approximate the contour with 4 points
    peri = cv.arcLength(cntr, True)
    frm = cv.approxPolyDP(cntr, 0.1*peri, True)
    if len(frm) != 4:
        continue

  # Converted image should fit into the original size
    board_size = max(imgw, imgh)
    if imgx + board_size >= w or imgy + board_size >= h:
        continue
  # Points should not be too close to each other
  # (use euclidian distance)
    if cv.norm(frm[0][0] - frm[1][0], cv.NORM_L2) < 0.1*peri or \
        cv.norm(frm[2][0] - frm[1][0], cv.NORM_L2) < 0.1*peri or \
        cv.norm(frm[3][0] - frm[1][0], cv.NORM_L2) < 0.1*peri or \
        cv.norm(frm[3][0] - frm[2][0], cv.NORM_L2) < 0.1*peri:
        continue
```

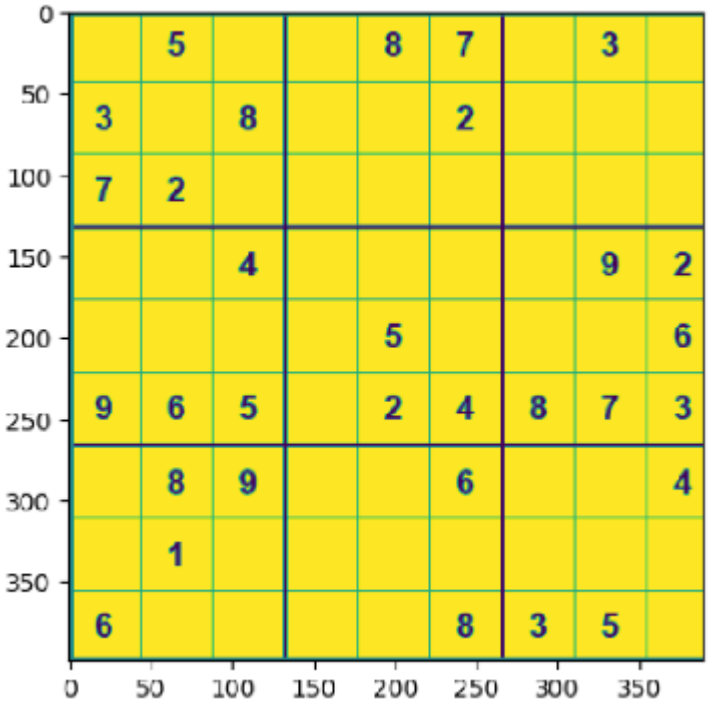
We iterate through each contour point until we identify 4 distinct points. To determine these points, we compute the Euclidean distance between each contour point. Our goal is to select points that are the farthest apart, resulting in the 4 most distant points. We then measure the contour's distance using its arclength and approximate the shape based on the identified arclength contours.



The Sudoku board contour

We crop the image based on the 4 points that compose the edges of the board.

```
croped_image = img_out[frm[0][0][1]:frm[1][0][1],frm[1][0][0]:frm[2][0][1]]
croped_image = cv.cvtColor(croped_image, cv.COLOR_BGR2GRAY)
```

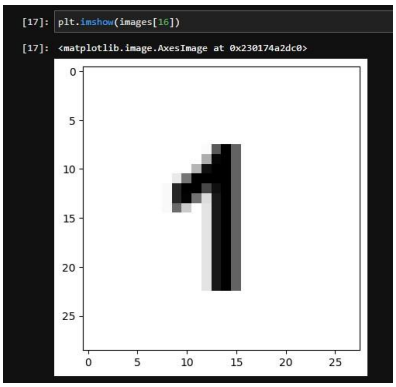


Cropped image

We now have the Sudoku board. Knowing that there are 81 squares of uniform size, we iterate over the board to crop each individual square.

```
images = []
cell_w, cell_h = imagenprueba.shape[1]//9,imagenprueba.shape[0]//9
for x in range(9):
    for y in range(9):
        x1, y1 = x*cell_w, y*cell_h
        x2, y2 = (x + 1)*cell_w, (y + 1)*cell_h
        cx, cy = (x1 + x2)//2, (y1 + y2)//2
        w2, h2 = cell_w, cell_h
        crop = imagenprueba[y1+10:y2-5, x1+10:x2-5]
        images.append(np.concatenate((np.expand_dims(crop, axis = 2),np.expand_dims(crop, axis = 2),np.expand_dims(crop, axis = 2)),axis = 2))
```

I noted discrepancies in the pixel sums and differences across individual squares. This adjustment was necessary to correct variations that resulted in distorted images of each square.



A picture of an individual square.

Second step: OCR to identify digits in each individual Square

This step is relatively straightforward. We feed each square image into an OCR tool to recognize the digit, subsequently appending the identified digit to a vector. For this task, I utilized EasyOCR, available at [GitHub - EasyOCR].

```
def get_digits(images):
    "get digit as string from images list and append them as integers into another list using an OCR"
    digits = []
    SudokuVector = []
    for i in range (81):
        print(f"Getting digit {i + 1} of 81")
        up_width = 400
        up_height = 400
        up_points = (up_width, up_height)
        resized_up = cv.resize(images[i], up_points, interpolation= cv.INTER_LINEAR)
        cv.imwrite("ActualPicture.JPG",resized_up)
        #Use OCR to extract digit from digit image
        result = reader.readtext('ActualPicture.JPG')



        if len(result)> 0:
            digits.append(int(result[0][1]))
        else:
            digits.append(0)

    #remove temporary JPG

    os.remove("ActualPicture.JPG")

    #transform digits list to the sudoku vector for backtracking algorithm

    for i in range (9):
        for j in range (9):
            SudokuVector.append(digits[(j * 9) + i])

    return SudokuVector
```

In the section marked with a red arrow, I resize the individual images. This adjustment became necessary as the OCR struggled to accurately identify digits when presented with the original image dimensions. It's important to mention that any white spaces are substituted with a '0' to facilitate the subsequent backtracking algorithm.

For the section indicated by the blue arrow, I sequentially add each digit to the aforementioned vector. Since the images are presented vertically, I implemented additional logic to arrange them in a horizontal sequence.

Third step: Backtracking Algorithm:

```
[12]: vector

[12]: [3,
       9,
       1,
       5,
       6,
       1,
       0,
       4,
       7,
       8,
       7,
       0,
       9,
```

Vector of a given Sudoku

```python
def backtrackingSudoku (vector,vector0,step,soluciones):
    if ValidateNumber(step,vector) == False:
        backtrackingSudoku (vector,vector0,step + 1,soluciones)
    elif ValidateNumber(step,vector) == True:
        for i in range (1,10):
            if ValidateRow(i,step,vector) and  ValidateColumn(i,step,vector) and Validate_box(i,step,vector):
                vector[step] = i
                if 0 not in vector:
                    if solucion(vector):
                        soluciones.append(vector.copy())
                else:
                    backtrackingSudoku (vector,vector0,step + 1,soluciones)
                vector[step] = 0
```

Backtracking Algorithm

The algorithm operates on a straightforward principle. The Sudoku puzzle is converted into a vector, maintaining a horizontal order. In this vector representation, a '0' indicates an empty space available for filling, while any other number represents a pre-filled value from the original puzzle.

During each iteration of the backtracking process, we advance to the subsequent index of the vector. If **Vector[step] != 0**, the algorithm recursively calls itself. However, when encountering a '0' at **Vector[step]**, we attempt to place digits ranging from 1 to 9. Before assigning a number, we ensure its validity by checking its presence in the corresponding row, column, and quadrant (the 9x9 box).

To ascertain whether a number can be placed in a specific position, we utilize multiples of 3 and 9 as indices for verification. For instance, when assessing **Vector[step]**, we examine if all elements **Vector[x * 9]** contain the desired number, thereby validating the column.

```
[10]: print("Original Sudoku:", end = "\n")
      printSudoku(vector0)
      print("Solution for Sudoku:", end = "\n")
      for i in soluciones:
          printSudoku(i)
          print(solucion(i))
```

```
Original Sudoku:
-------------------------------
[3, 9, 1] | [5, 6, 0] | [0, 4, 7]
[8, 7, 0] | [9, 4, 0] | [1, 2, 5]
[0, 0, 2] | [1, 8, 0] | [0, 0, 0]
-------------------------------
[0, 0, 0] | [2, 0, 9] | [0, 7, 8]
[9, 0, 0] | [0, 0, 4] | [6, 0, 1]
[0, 0, 3] | [7, 0, 0] | [0, 0, 0]
-------------------------------
[0, 3, 4] | [0, 0, 0] | [7, 0, 2]
[0, 0, 0] | [0, 0, 1] | [0, 6, 0]
[7, 6, 0] | [4, 0, 0] | [0, 0, 0]
-------------------------------
Solution for Sudoku:
-------------------------------
[3, 9, 1] | [5, 6, 2] | [8, 4, 7]
[8, 7, 6] | [9, 4, 3] | [1, 2, 5]
[5, 4, 2] | [1, 8, 7] | [3, 9, 6]
-------------------------------
[6, 1, 5] | [2, 3, 9] | [4, 7, 8]
[9, 2, 7] | [8, 5, 4] | [6, 3, 1]
[4, 8, 3] | [7, 1, 6] | [2, 5, 9]
-------------------------------
[1, 3, 4] | [6, 9, 5] | [7, 8, 2]
[2, 5, 8] | [3, 7, 1] | [9, 6, 4]
[7, 6, 9] | [4, 2, 8] | [5, 1, 3]
-------------------------------
True
```

Solved Sudoku

```
def printSudoku(vector):
    matrix =  get_matrix(vector)
    multiplesOf3= multiples(3,10)
    count = 0
    for i in multiplesOf3[:-1]:
        pre = i
        if (count) % 3 == 0:
            print("-"* 33)
        print(matrix[pre], "|", matrix[pre+1],"|", matrix[pre+2])
        count += 1
    print("-"* 33)
```

```
def ValidateNumber(step,vector):
    result = True
    if vector[step] != 0:
        result = False
    return result



def ValidateRow(number,step,vector):
    pre = 0
    result = True
    for i in multiples(9,10):
        if step > pre and step < i:
            if number in vector[pre:i]:
                result = False
        pre = i
    return result


def ValidateColumn(number,step,vector):
    result = True
    for i in multiples(9,9):
        if number == vector[step - i]:
            result = False
    return result
```

Example of validating Functions