# Extragerea informației  din careul Sudoku

Pentru rezolvarea celor două cerințe am dezvoltat o soluție pornind de la cele două script-uri prezentate în cadrul laboratorului de Computer Vision. Astfel, pentru ambele task-uri am utilizat funcțiile preprocess_image, normalize_image, show_image(pentru vizualizarea imaginilor procesate), precum si functia get_patch pentru a extrage patch-urile din imagine și  a stabili prezența/lipsa unei cifre.

## Task 1

### Procesarea imaginilor

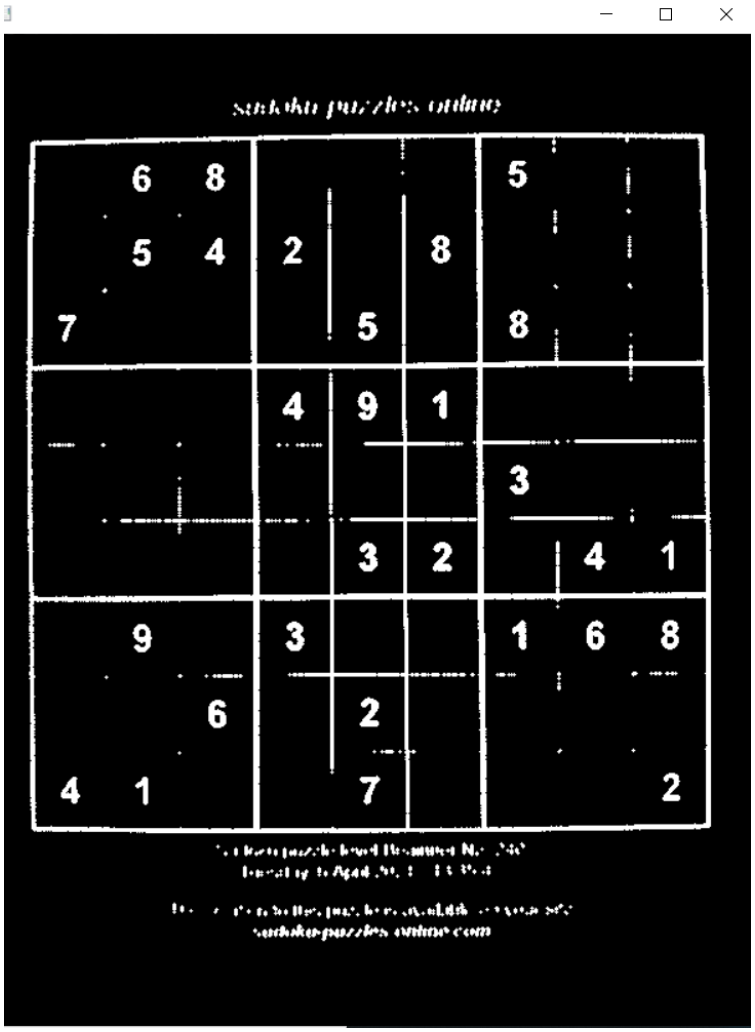
Soluția propusă pentru rezolvarea acestui task aduce toate imaginile la aceeași formă prin normalizarea și prelucrarea lor în metoda preprocess_image, care are ca parametri imaginea ce trebuie procesată și numărul taskului. Pentru a elimina detalii nefolositoare și evidențierea conturului am apelat inițial metoda turn_binary() unde se aplică pe imaginilie normalizate un filtru Gaussian Blur și  cv.addWeighted() pentru a stabili un contrast. Mai departe am transformat în binar totul prin aplicarea unui threshold adaptive însoțit de un dilate, deoarece liniile erau distorsionate. ceea ce nu ne ajuta în determinarea celui mai mare contur.

Al doilea pas a fost extragerea careului de Sudoku și rotirea imaginii astfel încât toate imaginile să fie asemănătoare unei matrice poziționate corect. Pentru acest lucru am ales să utilizez același algoritm din laborator pentru extragerea colțurilor. Mai exact, utilizand cv.findContours() din librăria openCV, iterăm prin toate contururile și îl determinăm pe cel de arie maximă, apoi găsim colțurile cele mai din stânga(sus/jos), respectiv dreapta și le returnăm.

```
def turn_binary(img):
    image_raw = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    kernel = np.array([[0., 1., 0.], [1., 1., 1.], [0., 1., 0.]], np.uint8)
    image = normalize_image(image_raw)
    image_g_blur = cv.GaussianBlur(image, (5, 5), 0)
    image_sharpened = cv.addWeighted(image, 1.2, image_g_blur, 1., 0)
    thresh = cv.adaptiveThreshold(image_sharpened, 255, cv.ADAPTIVE_THRESH_MEAN_C, \
                                  cv.THRESH_BINARY_INV, 3, 2)

    thresh = cv.dilate(thresh, kernel, iterations = 1)
    return thresh
def turn_binary_colored(img):
```

**Output turn_binary()**



# Extragerea careului

Colțurile preluate vor fi utilizate ulterior în metoda cut_sudoku(img, result) ce primește ca parametri imaginea și array-ul de coordonate. Pentru a ne reasigura că punctele se află în ordine (stânga sus, dreapta sus, stânga jos, dreapta jos) le ordonăm încă o dată și calculăm matematic cea mai mare distanță dintre două puncte pentru a extrage lățimea careului. Punctele vor corespunde acum noilor colțuri determinate, exprimate în funcție de lățime și înălțime(egale între ele):

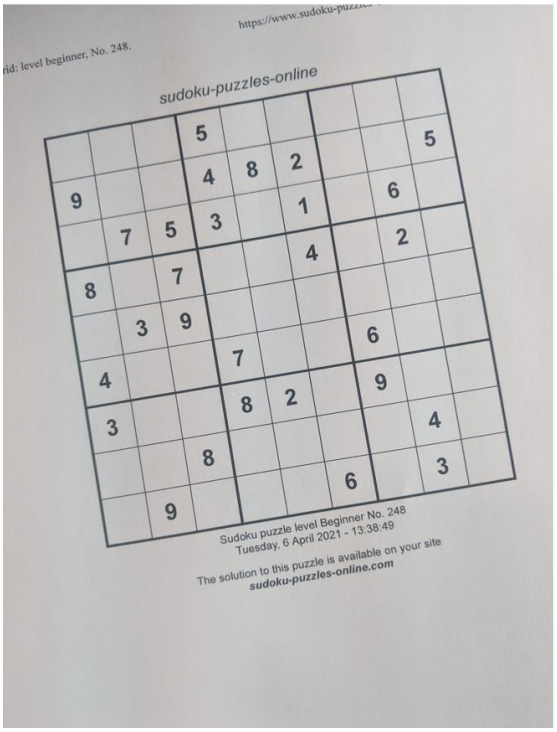
final_edge_points = np.array([[0, 0], [area - 1, 0], [0, area - 1], [area - 1, area - 1]], dtype="float32").

Având noua dimensiune a careului vom transforma imaginea comparând vechile coordonate cu noile coordonate și vom tăia, roti și aplatiza imaginea utilizând cv.warpPerspective(). Punctul cheie al acestei etape de extragere a fost reordonarea punctelor și determinarea noii dimensiuni a imaginii.

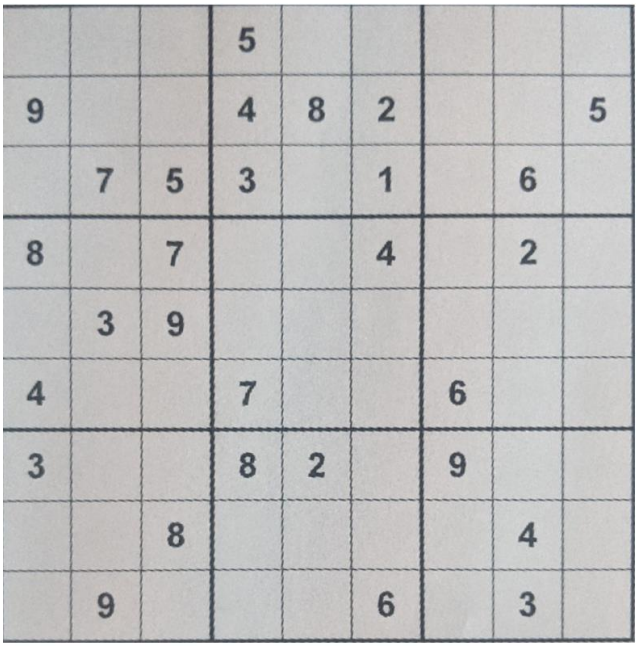
```python
def area_length(coord_x, coord_y):
    return math.sqrt((coord_x[0] - coord_y[0])**2 + (coord_x[1] - coord_y[1])**2)
def cut_sudoku(img, corners):
    edge_points = np.array(corners, dtype='float32')
    edge_points = np.float32(reorder(edge_points))
    area = max([
        area_length(corners[2], corners[1]),
        area_length(corners[0], corners[3]),
        area_length(corners[2], corners[3]),
        area_length(corners[0], corners[1])
    ])
    final_edge_points = np.array([[0, 0], [area - 1, 0], [0, area - 1], [area - 1, area - 1]], dtype="float32")
    matrix = cv.getPerspectiveTransform(edge_points, final_edge_points)
    return cv.warpPerspective(img, matrix, (int(area), int(area)))
```

**Output cut_sudoku(img, corners):**

**Înainte:**




**După:**



## Extragerea patch-urilor și identificarea cifrelor

Pentru identificarea cifrelor dintr-o casetă am ales să merg din patch în patch pe imaginea tăiată și rotită a careului și să compar media pixelilor. Pentru fiecare patch am restrâns dimensiunile acestuia, deoarece centrul lui este suficient pentru identificarea unei cifre. Fiind o imagine alb-negru, calculăm media pixelilor din caseta respectivă. Dacă aceasta este diferită de 0 (dacă există pixeli albi în casetă) atunci vom scrie în fisier “x”, în caz contrar “o”. În unele cazuri, deoarece algoritmul nu identifică ultimele linii din careu, apar erori în predicții, iar ultima linie și coloană riscă să nu fie analizate. Am rezolvat aceasta problemă prin implementarea unor condiții. În total matricea ar trebui să prezinte 10 linii verticale și 10 orizontale. În cazul în care nu se întamplă acest lucru vom identifica următoarele căsuțe folosind digit_weight și digit_height. Varibilele reprezinta dimensiunile unei casete din matrice calculate prin împărțirea lățimii și înălțimii la 9 (matrice de dimensiune 9x9).

```
def get_patch(f, img, lines_horizontal, lines_vertical, ls, digit_width, digit_height):
    len_vertical = len(lines_vertical)
    for i in range(len(lines_horizontal) - 1):
        for j in range(len(lines_vertical) - 1):
            y_min = lines_vertical[j][0][0] + 15
            y_max = lines_vertical[j + 1][1][0] - 10
            x_min = lines_horizontal[i][0][1] + 15
            x_max = lines_horizontal[i + 1][1][1] - 10
            patch = img[x_min:x_max, y_min:y_max].copy()
            mean_pixels = np.mean(patch)
            if mean_pixels != 0:
                f.write("x")
            else:
                f.write("o")
        if len(lines_vertical) == 9:
            len_vertical = 9
            y_min = lines_vertical[8][0][0] + 15
            y_max = lines_vertical[8][1][0] + digit_width - 10
            x_min = lines_horizontal[i][0][1] + 15
            x_max = lines_horizontal[i + 1][1][1] - 10
            patch = img[x_min:x_max, y_min:y_max].copy()
            mean_pixels = np.mean(patch)
            if mean_pixels != 0:
                f.write("x")
            else:
                f.write("o")
        if i != len(lines_horizontal) - 2:
            f.write("\n")
```

```
if len(lines_horizontal) == 9:
    f.write("\n")
    for j in range(len(lines_vertical) - 1):
        y_min = lines_vertical[j][0][0] + 15
        y_max = lines_vertical[j + 1][1][0] - 10
        x_min = lines_horizontal[len(lines_horizontal) - 1][0][1] + 15
        x_max = lines_horizontal[len(lines_horizontal) - 1][1][1] + digit_height - 10
        patch = img[x_min:x_max, y_min:y_max].copy()
        mean_pixels = np.mean(patch)
        if mean_pixels != 0:
            f.write("x")
        else:
            f.write("o")
    if len(lines_vertical) == 9:
        y_min = lines_vertical[len_vertical - 1][0][0] + 15
        y_max = lines_vertical[len_vertical - 1][1][0]+ digit_width - 10
        x_min = lines_horizontal[len(lines_horizontal) - 1][0][1] + 15
        x_max = lines_horizontal[len(lines_horizontal) - 1][1][1] + digit_height - 10
        patch = img[x_min:x_max, y_min:y_max].copy()
        mean_pixels = np.mean(patch)
        if mean_pixels != 0:
            f.write("x")
        else:
            f.write("o")
```

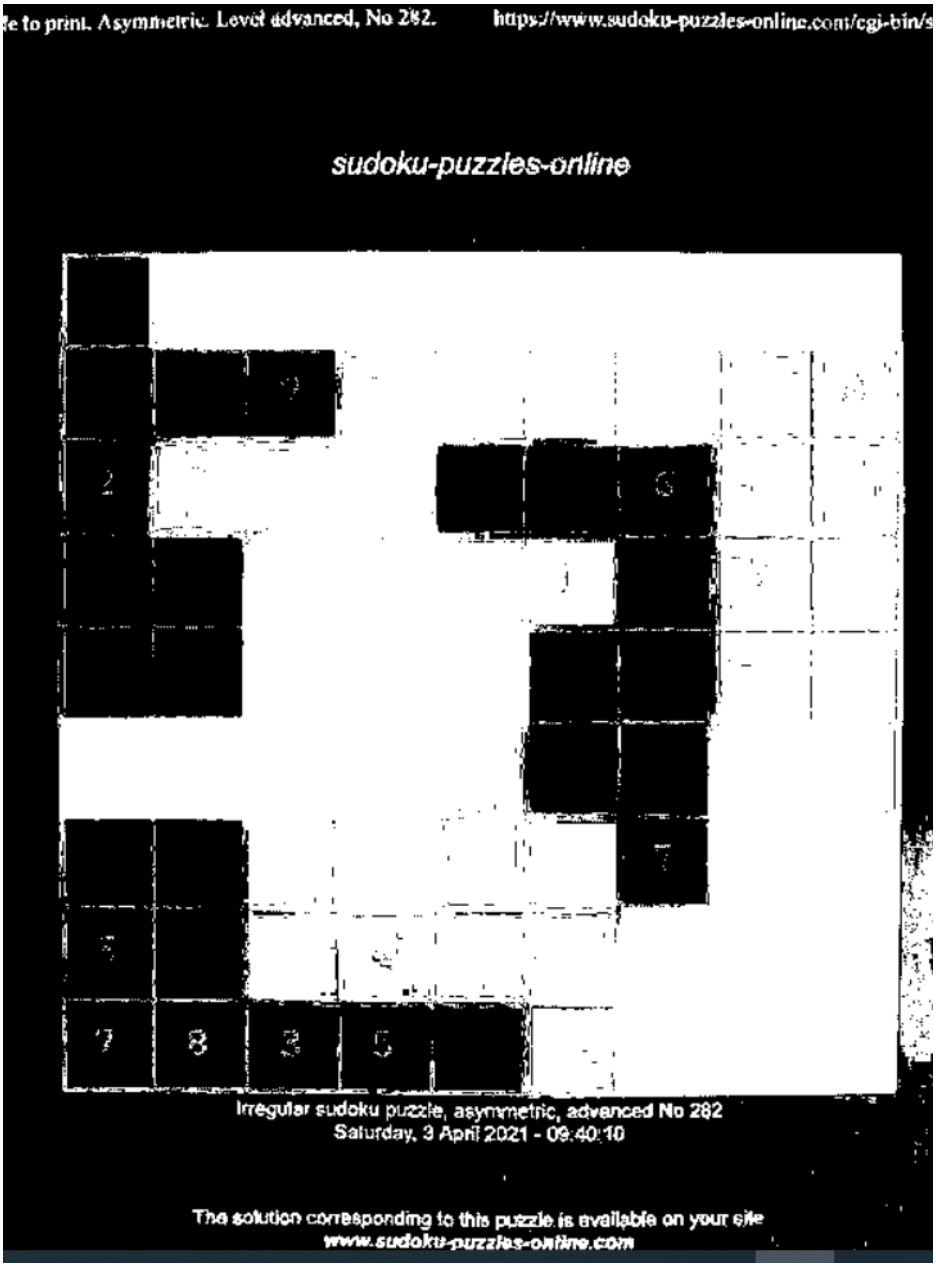
# Task 2

## Procesarea imaginilor

Soluția se deosebește de rezolvarea primului task prin determinarea paletei de culori pentru fiecare imagine în parte. După ce imaginea este citită din fișier programul verfică existența pixelilor roșii, deoarece Sudoku Jigsaw colorat conține, în mod cert, culorile roșu, galben și albastru pentru zone. Am setat pragurile de low și high în asa fel încât să obținem o imagine doar în negru și alb. Pentru a nu confunda imaginea nou creată cu cele deja alb-negru am calculat media pixelilor albi și media pixelilor negri. Făcând mai multe teste am observat că în cazul imaginilor colorate diferența dintre pixelii negri și pixelii albi este mul mai mică față de cea alb negru, așadar funcția va returna 1 în acest caz și 0 în caz contrar.

```
def color_filter(img_raw):
    img = cv.cvtColor(img_raw, cv.COLOR_BGR2HSV)
    red_index = np.array([30, 20, 1])
    red_last_index = np.array([255, 255, 255])
    grid = cv.inRange(img, red_index, red_last_index)
    if np.mean(grid == 0) - np.mean(grid == 255) >= 0.7:
        return 0
    return 1
```

Imagine filtrată:



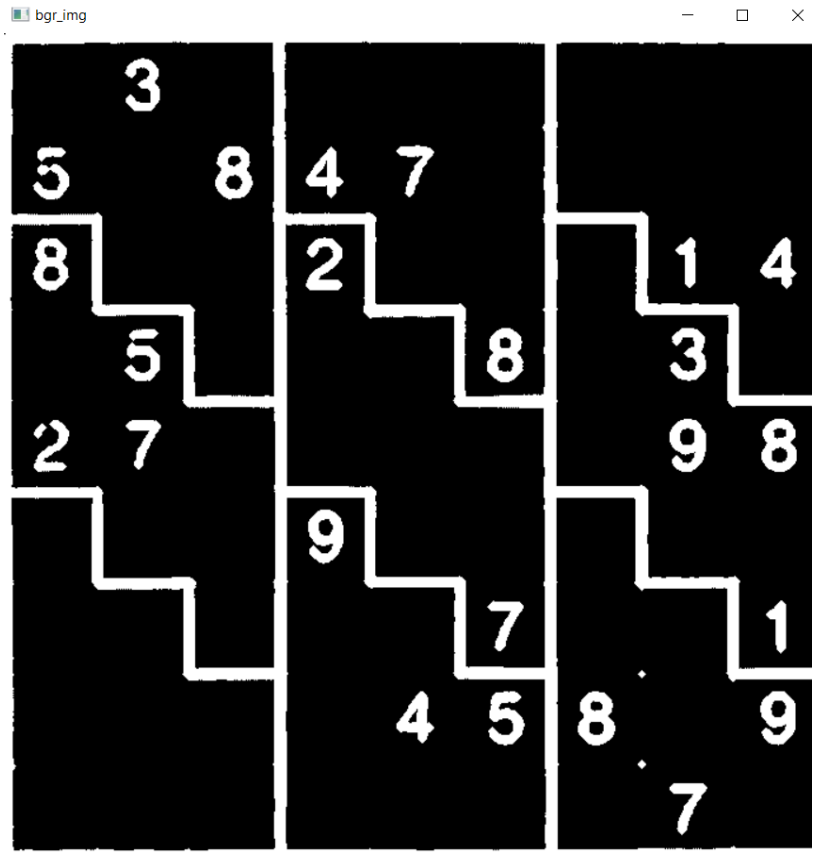

În cazul în care imaginea este colorată vom utiliza alte valori pentru procesarea imaginilor, precum și creșterea luminozității prin metoda:

```
average = 50
image_raw = np.where((255 - image_raw) < average, 255, image_raw + average)
```

În funcția turn_binary_colored().

Pentru pasul următor, și anume scoaterea careului, vom utiliza aceeași metodă ca la task-ul 1. Pentru extragerea zonelor, în schimb, în cazul imaginilor colorate, ne vom folosi tot de ideea filtrării culorilor. Așadar, vom proceda ca în funcția turn_binary_colored(), dar vom aplica procedeul pe fiecare culoare. Astfel, vom obține imagini cu zonele de culoare roșie evidențiate în alb și fundal negru, imagini cu zonele de culoare galbenă evidențiate în alb și fundal negru și imagini cu zonele de culoare albastră evidențiate în alb și fundal negru. Vom avea 3 seturi de imagini în final care vor conține, pe rând, zonele de fiecare culoare transformată în alb și fundal negru. În cazul în care imaginea este alb negru atunci o vom transforma în aceeași funcție într-o imagine alb-negru care evidențiază doar contururile îngroșate.

```
    return thresh
def get_colored_thresh(img_raw, color):
    img = cv.cvtColor(img_raw, cv.COLOR_BGR2HSV)
    if color == "gray":
        first_index = np.array([15, 15, 15])
        last_index = np.array([255, 255, 255])
    elif color == "green&red":
        first_index = np.array([0, 10, 50])
        last_index = np.array([70, 255, 255])
    elif color == "red":
        first_index = np.array([0, 10, 20])
        last_index = np.array([20, 255, 255])
    elif color == "green":
        first_index = np.array([20, 10, 0])
        last_index = np.array([70, 255, 255])
    elif color == "blue":
        first_index = np.array([60, 0, 20])
        last_index = np.array([200, 255, 255])
    grid = cv.inRange(img, first_index, last_index)
    return grid
```




*Imagine Jigsaw gri transformată*

După ce am obținut cele 3 imagini cu zonele respective vom reutiliza funcția get_patch() de la primul task și vom merge din patch în patch, simultan, pe cele 4 imagini: cea alb-negru, cea cu zonele roșii, cea cu zonele albastre și cea cu zonele galbene. Vom verifica pentru fiecare patch cărei zone aparține punând condiția ca media pixelilor albi să fie mai mare decât cea a pixelilor negri. În cazul în care patch-ul nu mai aparține zonei anterioare (I.e avem un patch total negru) asta înseamnă că am întalnit o nouă zonă. Pentru a putea ține evidența acestor zone am creat o nouă imagine de aceleași dimensiuni cu imaginea procesată și am inițializat-o cu 0. Atfel, de fiecare dată când întâlnim o nouă zonă căutăm toate patch-urile care aparțin acelei zone(sunt conectați direct între ei și conțin pixeli albi) și vom marca toți pixelii din acele patch-uri cu numărul zonei respective.

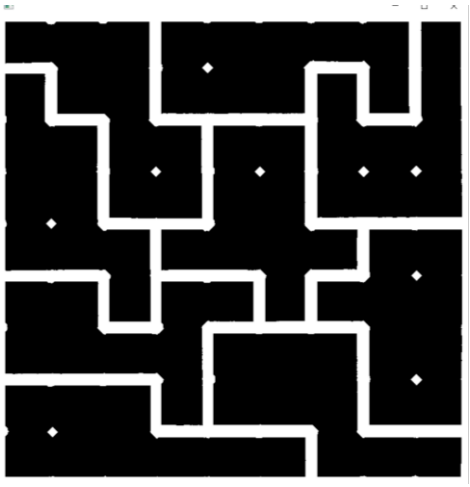
```
    for j in range(len(lines_vertical) - 1):
        y_min = lines_vertical[j][0][0] + 20
        y_max = lines_vertical[j + 1][1][0] - 20
        x_min = lines_horizontal[i][0][1] + 20
        x_max = lines_horizontal[i + 1][1][1] - 20
        patch = img[x_min:x_max, y_min:y_max].copy()
        patch_blue = thresh_blue[x_min:x_max, y_min:y_max].copy()
        patch_green = thresh_green[x_min:x_max, y_min:y_max].copy()
        patch_red = thresh_red[x_min:x_max, y_min:y_max].copy()
        mean_pixels_black_red = np.mean(patch_red == 0)
        mean_pixels_white_red = np.mean(patch_red == 255)
        mean_pixels_black_green = np.mean(patch_green == 0)
        mean_pixels_white_green = np.mean(patch_green == 255)
        mean_pixels_black_blue = np.mean(patch_blue == 0)
        mean_pixels_white_blue = np.mean(patch_blue == 255)
        #If we reached a new zone, then label each patch of it
        if mask[x_min:x_max, y_min:y_max].all() == 0:
            if mean_pixels_white_red > mean_pixels_black_red:
                mask, zone = get_depth(thresh_red, mask, zone, lines_horizontal, lines_vertical,y_min - 20, x_min - 20)
            elif mean_pixels_white_blue > mean_pixels_black_blue:
                mask, zone = get_depth(thresh_blue, mask, zone, lines_horizontal, lines_vertical,y_min - 20, x_min - 20)
            elif mean_pixels_white_green > mean_pixels_black_green:
                mask, zone = get_depth(thresh_green, mask, zone, lines_horizontal, lines_vertical,y_min - 20, x_min - 20)
        mean_pixels = np.mean(patch)
        empty = ""
        number_zone = 0
        if mean_pixels != 0:
            empty = "x"
        else:
            empty = "o"
        for k in range(x_min,x_max):
et_colored_patch() › for i in range(len(lines_horizo... › for j in range(len(lines_vertic...
```

Pentru identificare și marcarea unei zone noi avem nevoie, în mod evident, ca toți pixelii dintr-un patch să fie marcați cu 0, adica nevizitați. Pentru modificare acestui lucru și incrementarea zonei am utilizat funcția get_depth() care primește imaginea zonei din care face parte, numărul zonei, liniile veticale si orizontale și coordonatele de start, deoarece ar fi ineficientă căutarea aceluiași patch de la începutul careului.

```
def get_depth(img,mask, zone_label,lines_horizontal, lines_vertical, start_width,start_height):
    height, width = img.shape[0], img.shape[1]
    visited = np.zeros((height, width))
    digit_box_width = width // 9
    digit_box_height = height // 9
    next_visited = Queue()
    next_visited .put([start_height, start_width])
    counter = 0
    while next_visited .empty() == False:
        (coord_x, coord_y) = next_visited .get()
        i, j = coord_x, coord_y
        # If there is a white patch from the current zone then search its neighbors
        # Otherwise, we are outside the are
        if visited[i, j] == 0 and mask[i, j] == 0 and \
                np.mean(img[i:i+digit_box_height,j:j+digit_box_width] == 255) > np.mean(img[i:i+digit_box_height,j:j+digit_box_width] == 0):
            mask[i:i+digit_box_width, j:j+digit_box_height] = zone_label
            visited[i:i+digit_box_width, j:j+digit_box_height] = 1
            counter +=1
            neighbors = []
```

```
            if coord_x + 2*digit_box_height <= height:
                patch = img[coord_x + digit_box_height:coord_x + 2*digit_box_height, coord_y:coord_y+digit_box_width].copy()
                if np.mean(patch == 255) > np.mean(patch == 0):
                    neighbors.append([coord_x + digit_box_height, coord_y])
            if coord_x - digit_box_height >= 0:
                patch = img[coord_x - digit_box_height:coord_x, coord_y:coord_y + digit_box_width].copy()
                if np.mean(patch == 255) > np.mean(patch == 0):
                    neighbors.append([coord_x - digit_box_height, coord_y])
            if coord_y + 2*digit_box_width <= width:
                patch = img[coord_x:coord_x+digit_box_height, coord_y + digit_box_width:coord_y + 2*digit_box_width].copy()
                if np.mean(patch == 255) > np.mean(patch == 0):
                    neighbors.append([coord_x, coord_y + digit_box_width])
            if coord_y - digit_box_width >= 0:
                patch = img[coord_x:coord_x+digit_box_height, coord_y - digit_box_width:coord_y].copy()
                if np.mean(patch == 255) > np.mean(patch == 0):
                    neighbors.append([coord_x, coord_y - digit_box_width])
            for neighbor in neighbors:
                if visited[neighbor[0],neighbor[1]] == 0 and mask[neighbor[0],neighbor[1]] == 0:
                    next_visited .put(neighbor)
    if counter != 0:
        zone_label += 1
    return mask, zone_label
```

În cazul extragerii zonelor dintr-o imagine gri de Sudoku Jigsaw  am folosit același principiu și anume, extragerea unei imagini pentru aproximativ fiecare zonă în parte. Dat fiind faptul că imaginile gri scot în evidență foarte bine conturul zonelor după prelucrare am aplicat cv.findContours() pentru a gasi contururile de lungime cea mai maredin imagine. Înainte de asta, cum imaginea este binară, am șters cu funcția clear_patches() fiecare patch care conținea o cifră schimbându-i toți pixelii albi în pixeli negri.

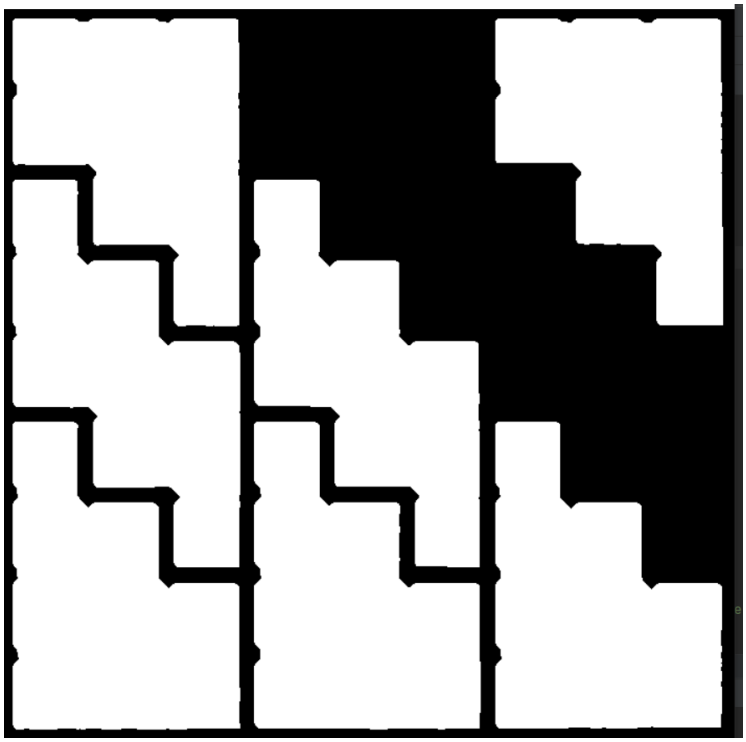


*Imaginea după aplicarea clear_patches()*

Apoi am extras cele mai mari contururi și pentru fiecare contur în parte am salvat o imagine în care aria acestuia este înlocuită de pixeli albi, imagine pe care am adăugat-o în array-ul single_contours. În același timp evidențiem zonele pe o singură imagine, obținându-le pe toate cu excepția ultimei. Pentru a rezolva acest lucru am aplicat un dilate pe imaginea tuturor contururilor pentru a elimina orice linii existente și a rămâne doar cu ultima zonă pe care la final o tranformăm în pixeli albi:

```
new_image = cv.bitwise_not(all_contours)
```

Procesul nu extrage întotdeauna cele 9 zone, considerând două sau mai multe zone drept una singură. Așadar pentru zonele respective voi mai aplica înco o data findContours() și voi găsi zonele care au fost unite. Înainte de de acest procedeu vom mai aplica cv.erode() pentru eliminare noise-ului și punerea în evidență a conturului inițial.



*Exemplu de două zone considerate una singură*

```
all_contours = cv.dilate(all_contours, kernel, iterations=20)
kernel = np.array([[0., 1., 0.], [1., 1., 1.], [0., 1., 0.]], np.uint8)
new_image = cv.bitwise_not(all_contours)
last_zones = cv.bitwise_not(all_contours)
if len(single_contour) < 8:
    img = cv.erode(img, kernel, iterations = 3)
    new_image = cv.bitwise_not(img)
    new_image = cv.bitwise_and(last_zones, last_zones, mask = new_image)
    new_image = cv.dilate(new_image, kernel, iterations = 2)
    edges = cv.Canny(new_image, 255, 400)
    contours, _ = cv.findContours(edges, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
    for i in range(len(contours)):
        if len(contours[i]) > 3:
            mask = np.zeros(img.shape, np.uint8)
            cv.drawContours(mask, [contours[i]], 0, (255, 255, 255), -1)
            single_contour.append(mask)
single_contour.append(new_image)
set_zones(f, img_copy, single_contour, lines_horizontal, lines_vertical)
```

La final, vom proceda asemănător ca la Sudoku Jigsaw colorat și vom seta zonele în funcție de imaginile determinate.