EXTRAGEREA INFORMAȚIEI VIZUALE DIN CAREURI SUDOKU

Soluția abordată poate fi descompusă în următorii pași:

1. procesarea imaginilor inițiale pentru a extrage din ele careul cu sudoku;
2. împărțirea careului sudoku în 81 de pătrate, care vor fi analizate pentru a furniza output-ul.

Pasul 1:

Pentru fiecare imagine dorim întâi să o rotim astfel încât să devină paralelă cu axele Ox și Oy, după care să extragem careul cu sudoku.

Pentru aceasta, modificăm imaginea astfel încât să fie formată doar din pixeli albi și negri, prin funcția *preprocess_image*:

```
# varianta pentru sudoku clasic; pentru jigsaw, vom schimba doar dimensiunea
# filtrului median la 5
image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
image_m_blur = cv.medianBlur(image, 3)
image_g_blur = cv.GaussianBlur(image_m_blur, (0, 0), 5)
image_sharpened = cv.addWeighted(image_m_blur, 1.2, image_g_blur, -0.8, 0)
_, thresh = cv.threshold(image_sharpened, 30, 255, cv.THRESH_BINARY)

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

edges = cv.Canny(thresh, 150, 400)
contours, _ = cv.findContours(edges, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
```

În continuare, parcurgem fiecare contur returnat de funcția findContours din opencv. Determinăm punctele extreme ale acestuia, iar dacă aria descrisă de ele este maximă, înseamnă că am găsit conturul exterior și salvăm punctele. Calculăm și punctul care reprezintă mijlocul careului, care va fi util la rotirea imaginii.

După ce obținem aceste date, calculăm diferențele de nivel pe cele 4 laturi, care ne vor fi utile la aflarea unghiurilor de înclinare:

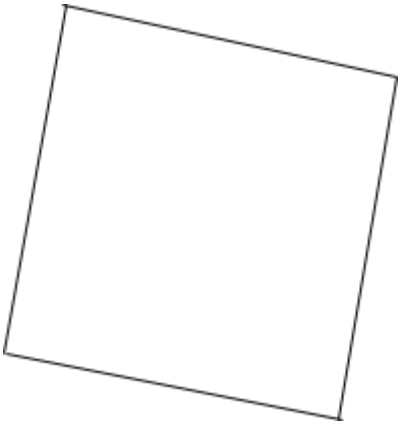
```
# calculez diferentele intre punctele de sus, de jos, din stanga si din dreapta
top_difference = abs(top_right - top_left)
left_difference = abs(top_left - bottom_left)
bottom_difference = abs(bottom_left - bottom_right)
right_difference = abs(bottom_right - top_right)

# calculez unghiurile de inclinare pe fiecare dintre cele 4 laturi
angle_top = np.arctan(top_difference[1] / top_difference[0])
angle_left = np.arctan(left_difference[0] / left_difference[1])
```

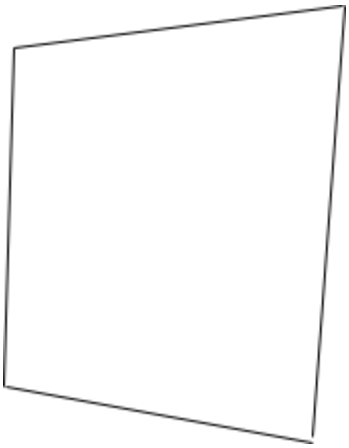
```
    angle_bottom = np.arctan(bottom_difference[1] / bottom_difference[0])
    angle_right = np.arctan(right_difference[0] / right_difference[1])
```

Unghiul cu care trebuie să rotim imaginea va fi media celor 4 valori calculate anterior, iar rezultatul obținut îl vom converti la grade din radiani.

Următorul pas este să rotim imaginea cu unghiul calculat, direcția de rotire fiind dată de înclinarea laturii de sus. În exemplul de mai jos, trebuie să facem o rotire la stânga pentru a obține imaginea dreaptă.



Dar mai pot apărea și excepții, ca în exemplul acesta:



Deși înclinarea laturii de sus ne sugerează că trebuie să efectuăm o rotire la dreapta, observăm că unghiul de jos(în acest caz) este mai mare decât cel de sus și ar fi mai bună o rotire la stânga.

După rotire, aflăm din nou colțurile careului și decupăm, cu ajutorul acestor coordonate, zona de interes.

Pasul 2:

Avem imagini care conțin doar sudoku, de dimensiune 500x500. Fiecare celulă dintre cele 81 va avea dimensiunile aproximative 55x55. Pentru a stabili dacă avem spațiu liber sau nu, din patch-ul inițial tăiem câte 15 pixeli pe fiecare dintre cele 4 laturi, pentru a elimina eventuale linii rămase din sudoku și analizăm zona centrală rămasă cu funcția *modify_patch*:

```
def modify_patch(image): # modific fiecare patch pana ajung la o imagine cu
    # pixeli albi si negri
    image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    image_m_blur = cv.medianBlur(image, 3)
    image_g_blur = cv.GaussianBlur(image_m_blur, (0, 0), 5)
    image_sharpened = cv.addWeighted(image_m_blur, 1.2, image_g_blur, -0.8, 0)
    _, thresh = cv.threshold(image_sharpened, 30, 255, cv.THRESH_BINARY)

    # pentru a verifica daca celula este goala, fac media imaginii; daca media
    # este 255, inseamna ca am doar pixeli albi
    patch_mean = np.mean(thresh)
    if patch_mean < 254:
        return 'x'
    else:
        return 'o'
```

Pentru sudoku jigsaw, mai trebuie să facem în plus detectarea și numerotarea regiunilor neregulate. Următoarea secvență de cod arată modificările aduse unei imagini cu sudoku pentru a pune în evidență liniile de separare dintre regiuni și localizarea acestora:

```
    # modific imaginea pentru a pune in evidenta liniile care separa grupurile
    image = cropped_images[idx - 1].copy()
    image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    image_m_blur = cv.medianBlur(image, 7)
    image_g_blur = cv.GaussianBlur(image_m_blur, (0, 0), 7)
    image_sharpened = cv.addWeighted(image_m_blur, 1.4, image_g_blur, -0.95, 0)
    _, thresh = cv.threshold(image_sharpened, 30, 255, cv.THRESH_BINARY)
    # obtin o imagine alb-negru

    # aplic dilatare pentru a ramane in evidenta muchiile de separare
    kernel = np.ones((3, 3), np.uint8)
    thresh = cv.dilate(thresh, kernel)

    # desenez contururile pe imagine
    contours, _ = cv.findContours(thresh, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE)
    new_img = cv.drawContours(thresh, contours, -1, color=(0, 255, 0), thickness=1)

    sep_between_lines = [[False for _ in range(9)] for _ in range(9)]
    # sep_between_lines[i][j] = True daca linia i este separata de linia i + 1 pe
    # coloana j; False altfel
    sep_between_columns = [[False for _ in range(9)] for _ in range(9)]
   # sep_between_columns[i][j] = True daca coloana j este separata de coloana j + 1
   # pe linia i; False altfel

    # fiecare posibila linie de separare are coordonate multiplii de 55 (imaginea are
    # h = w = 500, deci un patch are h = w = 55)
    for i in range(55, 55 * 8 + 1, 55):
        for j in range(0, 55 * 8 + 1, 55):
            for k in range(-15, 16): # k este o marja de erore de 15 pixeli (liniile
```

```
                # pot fi deviate, decuparea nu iese mereu perfect)
                s = np.sum(new_img[j: j + 55, i + k]) # calculez suma pixelilor de pe
                                                      # linia curenta
                if s <= 255 * 10: # daca suma are maxim 10 pixeli albi, inseamna ca
                # sunt minim 45 de pixeli negri, deci avem separare intre coloane
                    sep_between_columns[j // 55][i // 55 - 1] = True
                    break

            # analog determinam separarea intre linii
            for k in range(-15, 16):
                s = np.sum(new_img[i + k, j: j + 55])
                if s <= 255 * 10:
                    sep_between_lines[i // 55 - 1][j // 55] = True
                    break
```

Știind unde avem linii care separă grupurile, acum putem determina zonele exacte folosind algoritmul lui Lee, plecând dintr-o celulă care nu este adăugată încă într-un grup, la care adăugăm toți vecinii ei care nu au un grup și nu sunt separați de segmentele determinate anterior.

Pentru task-ul bonus, la determinarea cifrelor am folosit template matching (funcția matchTemplate din cv2, cu metoda de comparare cv2.TM_CCOEFF_NORMED), iar cifrele folosite drept template au fost decupate din imaginile de antrenare. Acestea se găsesc în folderul *cifre*, care se află în același director cu codul sursă.

Înainte de a verifica similaritatea, patch-urile și imaginile cu cifre au fost modificate prin funcția *modify_image*:

```
def modify_image(img): # modific imaginile astfel incat la final sa contina doar
    # pixeli albi si negri
    image = img.copy()
    image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    image_m_blur = cv.medianBlur(image, 3)
    image_g_blur = cv.GaussianBlur(image_m_blur, (0, 0), 5)
    image_sharpened = cv.addWeighted(image_m_blur, 1.2, image_g_blur, -0.8, 0)
    _, thresh = cv.threshold(image_sharpened, 30, 255, cv.THRESH_BINARY)

    return thresh
```

Pentru varianta clasică, la fiecare patch marcat cu ‘x’ am calculat cât de mult se aseamănă cu una dintre cele 9 cifre, rezultatul final fiind cifra cu scor maxim de similaritate.

Pentru varianta jigsaw, am avut 2 seturi de cifre (un set extras din sudoku-rile color, iar celălalt din sudoku-rile alb-negru). Pentru fiecare set, am aplicat metoda anterioară și am obținut 2 rezultate, dintre care l-am păstrat pe cel cu scor mai mare pentru a prezice cifra.