Incarcarea bibliotecilor Pentru acest proiect vom folosi numpy, matplotlib si opencv.

```
import numpy as np
import os
import cv2 as cv
```

Identificarea datelor constante

Traseul de scor Observam ca din moment ce tabla de joc este aceiasi pentru toate jocurile de DDD, chenarul ei, adica traseul de scor, ramane neschimbat. Astfel, putem sa retinem, intr-un dictionar, numarul de la 0 la 6 asociat fiecarei pozitii de scor

Punctajul de scor asociat coordonatelor din matricea jocului Deoarece tabla de joc nu se modifica, putem construi un dictionar cu ajutorul caruia sa asociem fiecarei pozitii cu puncte de pe tabla de joc numarul corespunzator de puncte. Dictionarul ar avea perechi key - value de forma (line, column) - points

Simplificarea experimentelor Pentru a construi mai usor masti pentru extragerea continutului din imagini, se foloseste trackbar-ul implementat la laborator. Cu ajutorul acestuia identificam o masca ce ne elimina piesele de domino pentru a calcula dimensiunea tablei de joc efective

De asemenea, folosim functia pentru afisarea imaginilor de la laborator, cu mici modeificari

Preprocesari Inainte de a intra efectiv in parcurgerea imaginilor si identificarea pieselor, vrem sa facem anumite calcule pentru a putea decupa din pozele viitoare doar tabla de joc. In acest sens, urmam urmatorii pasi:

- obtinem o masca pentru pastrarea exclusiv a tablei de joc
- inchidem imaginea rezultata pentru a astupa golurile din tabla
- deschidem imaginea rezultata pentru a elimina pionii
- identificam prin coordonatele colturilor stanga sus si dreapta jos tabla de joc

De ce facem asta? Pe baza acestor coordonate ale tablei de joc, vom putea decupa imaginile pe masura ce le procesam pentru a identifica strict tabla de joc. Acest lucru ne ajuta deoarece, la fiecare pas cand analizam diferentele dintre imagini nu vrem ca variatiile in luminozitatea mesei si traseului de scor sa conduca la delimitari gresite ale unei piese nou adaugate

Operatorii Morfologici Deoarece in cele ce urmeaza vom folosi de mai multe ori operatii de inchideri si deschideri, vom defini doua functii in acest sens

```
def get_open_image(source_image, kernel_size=2):
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    source_image = cv.erode(source_image, kernel, iterations=1)
    source_image = cv.dilate(source_image, kernel, iterations=1)

    return source_image

def get_closed_image(source_image, kernel_size=2):
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    source_image = cv.dilate(source_image, kernel, iterations=1)
```

```
    source_image = cv.erode(source_image, kernel, iterations=1)

    return source_image
```

Inchidere Umplem golurile din tabla de joc prin dilatare si eroziune

Deschidere Eliminam pionii din tabla de joc prin eroziune si dilatare

Identificam dimeniunea tablei de joc Pentru a calcula dimensiunea tablei de joc, ne trebuie coordonatele coltului din stanga sus si dreapta jos. O abordare naiva dar gresita ar fi sa parcurgem imaginea ca matrice si prima valoare ne-neagra sa fie coltul stanga sus iar ultima sa fie coltul dreapta jos.

Problema? Imaginea noastra nu este perfecta, mai exista la inceput si final linii incomplete De exemplu, daca ultima linie cu valori ne-negre din imagine are doar 3 pixeli, atunci algoritmul va identifica coltul dreapta jos ca fiind apropiat la nivel de coloana cu acel pixel din coltul de stanga sus, ceea ce strica iremediabil geometria ce se aplica la urmatorii pasi

Solutia? Construim o matrice caracterisitca pentru imaginea data, in care fiecare valoare ne-neagra are asociata suma coordonatelor sale Astfel, coltul real din stanga sus trebuie sa aiba valoarea minima, fiind cel mai apropiat de punctul (0, 0), in timp ce coltul real din dreapta jos trebuie sa aiba valorea maxima, fiind cel mai indepartat de punctul (0, 0) In acest sens definim functiile:

- **get_freq** care obtine o astfel de matrice caracterisitica pentru o imagine color data ca input
- **get_corner** care primeste ca input matricea caracteristica a unei imagini si un parametru pentru a specifica ce colt se doreste

```
def get_freq(img):
    freq = np.zeros((img.shape[0], img.shape[1]))

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if img[i][j][0] != 0 and img[i][j][1] != 0 and img[i][j]
[2] != 0:
                freq[i][j] = i + j

    return freq

def get_corner(freq, corner="LR"):
    if corner == "LR":
        max_value = freq.max()

        max_values_x = np.where(freq == max_value)[0]

        max_values_y = np.where(freq == max_value)[1]

        middle_value_x = max_values_x[len(max_values_x) // 2]

        middle_value_y = max_values_y[len(max_values_y) // 2]
```

```
        middle_value = (middle_value_x, middle_value_y)

        return middle_value
    elif corner == "UL":
        min_value = freq[freq > 0].min()

        min_values_x = np.where(freq == min_value)[0]

        min_values_y = np.where(freq == min_value)[1]

        middle_value_x = min_values_x[len(min_values_x) // 2]

        middle_value_y = min_values_y[len(min_values_y) // 2]

        middle_value = (middle_value_x, middle_value_y)

        return middle_value
```

Aplicam functiile pentru a obtine, din imaginea initiala, coordonatele pixelilor ce delimiteaza tabla de joc efectiva

Pentru verificare si vizualizare coloram vecinatatea lor cu rosu si observam ca sunt bine alesi Deoarece vom mai face aceasta verificare ulterior, o punem intr-o functie

Wrapuim aceasta logica de obtinere a colturilor tablei intr-o functie, cu mentiunea ca vom considera un decalaj de +/- 1% pentru cele doua colturi ca sa evitam situatiile in care pe tabla este pusa o piesa usor inclinata si prin decuparea din imaginea initiala a regiunii cu tabla de joc aceasta piesa nu mai este "vazuta" in intregime, conducand la misclasificare. Aceasta este o optimizare ulterioara adusa celor doua colturi intrucat problema a fost sesizata in etapa de testare

```
def get_table_corners():
    image_path = "imagini_auxiliare/01.jpg"
    image_start = cv.imread(image_path)
    image_start = cv.resize(image_start, (0, 0), fx=0.4, fy=0.4)

    image_start_hsv = cv.cvtColor(image_start, cv.COLOR_BGR2HSV)

    # Valorile experimentale sunt H: 68-255, S: 158-255, V: 139-255
    lower = np.array([68, 158, 139])
    upper = np.array([255, 255, 255])
    mask_hsv = cv.inRange(image_start_hsv, lower, upper)
    image_start_hsv = cv.bitwise_and(image_start, image_start,
mask=mask_hsv)
    new_image_start = cv.cvtColor(image_start_hsv, cv.COLOR_BGR2RGB)

    new_image_start = get_closed_image(new_image_start, 64)

    new_image_start = get_open_image(new_image_start, 64)
```

```
    freq = get_freq(new_image_start)

    first_pixel = get_corner(freq, "UL")
    last_pixel = get_corner(freq, "LR")

    first_pixel = (first_pixel[0] - int(0.01 *
new_image_start.shape[1]),
                   first_pixel[1] - int(0.01 *
new_image_start.shape[0]))

    last_pixel = (last_pixel[0] + int(0.01 *
new_image_start.shape[1]),
                    last_pixel[1] + int(0.01 *
new_image_start.shape[0]))

    return first_pixel, last_pixel
```

Transformarea imaginilor Ne dorim ca la fiecare pas sa avem o metoda de a identifica ce piesa de domino s-a adaugat, cu alte cuvinte ce s-a schimbat de la imaginea precedenta. Solutia este sa scadem cele doua imagini si sa tratam diferenta lor ca fiind o imagine ce contine, **IN TEORIE**, doar o piesa de domino pe care sa o putem localiza prin colturile stanga sus si dreapta jos ca apoi sa o putem decupa si procesa

Problema? Doar in teorie diferenta este strict piesa de domino. In practica, mici diferente in orientare si luminozitate conduc la mai multe modificari ce altereaza forma piesei depistate

Solutia? Identificam experimental o masca prin care putem sa ramanem, pe cat se poate, doar cu piesele de domino dintr-o imagine.

In cele ce urmeaza vom identifica aceasta masca pe baza imaginii cu domino-uri verticale

Experimentam cu imaginea si masca si obtinem valorile experimentale

Deoarece vom face aceasta operatie de mai multe ori, o vom wrapui intr-o functie

```
def get_dominoes_image(source_bgr_image):
    source_hsv_image = cv.cvtColor(source_bgr_image, cv.COLOR_BGR2HSV)

    # Valorile experimentale sunt H: 83-255, S: 0-97, V: 222-255
    lower = np.array([83, 0, 222])
    upper = np.array([255, 97, 255])
    mask_hsv = cv.inRange(source_hsv_image, lower, upper)
    source_hsv_image = cv.bitwise_and(source_bgr_image,
source_bgr_image, mask=mask_hsv)
    new_source_image = cv.cvtColor(source_hsv_image, cv.COLOR_BGR2RGB)

    return new_source_image
```

Simulam primul pas Inainte de a ne aventura in succesiunea de imagini vom testa pe primul pas: imaginea initiala versus prima imagine Citim prima imagine de joc, adica cea cu prima piesa

Aplicam masca pentru domino-uri pe imaginea de start

Observam ca au mai ramas evidentiate si zone care nu sunt piese de domino, stelutele cu puncte si logo-ul Aplicam operatia de deschidere pentru a ne apropia cat mai mult de o tabla neagra in mijloc, insa fara a altera prea tare piesele de domino ca sa putem sa mai identificam dupa aceea. Daca am eroda prea tare initial, chiar daca dilatam apoi, golurile negre din piesele de zar s-ar mari atat de mult incat s-ar strica piesa si nu o vom mai putea recunoaste ulterior, asa ca ne uitam cu atentie la dominourile de pe margine cat sa fie identificabile usor macar cu ochiul nostru de om

Aplicam masca de domino-uri si pe imaginea de joc si deschidem. Deoarece se fac aceleasi operatii ca mai sus, vom defini o functie care porneste de la imaginea citita si returneaza imaginea cu masca si deschisa

Identificarea diferentei dintre imagini In cele ce urmeaza incepe adevarata parte de computer vision a proiectului.

Deoarece vom face aceasta operatie la fiecare pas, o wrapuim intr-o functie. Experimental si modificarea design-ului arhitecturii de procesare a imaginii, s-a ajuns la urmatoare modalitate de a trata diferenta dintre cele doua imagini:

- Calculam diferenta directa dintre cele doua imagini
- Concertim rezultatul la tonuri de gri
- Binarizam imaginea aplicand un prag pentru intesitati
- Pastram din imagine doar regiunea cu tabla de joc
- Redimensionam imaginea ca sa aiba mai multe detalii
- Binarizam iar cu acelasi prag

```
def get_difference (current_image, previous_image, first_pixel,
last_pixel):
    difference = cv.subtract(current_image, previous_image)
    difference = cv.cvtColor(difference, cv.COLOR_BGR2GRAY)
    difference = cv.threshold(difference, 200, 255, cv.THRESH_BINARY)
[1]
    difference = difference[first_pixel[0]:last_pixel[0],
first_pixel[1]:last_pixel[1]]
    difference = cv.resize(difference, (0, 0), fx=3, fy=3)
    difference = cv.threshold(difference, 200, 255, cv.THRESH_BINARY)
[1]

    return difference
```

Clasificarea pieselor de domino In acest moment, ce trebuie sa facem pentru fiecare jumatate de domino e sa stabilim daca e 0, 1, 2, 3, 4, 5, 6 sau 7 Chiar daca problema s-ar preta bine pe un model de CNN studiat la Inteligenta artificala, sarcina noastra nu e sa deducem pentru piese de domino proaspete, nemaivazute, ce numar reprezinta, nu vrem neaparat o solutie generala Astfel, pe cazul nostru, practic ce vrem sa facem e sa comparam fiecare piesa cu piese deja cunoscute de zar si sa vedem care se potriveste cel mai bine In acest sens, ce ne rezolva problema este mecanismul de **pattern matching** din OpenCV

Ce ne dorim? Vrem ca pentru o imagine si un template sa stabilim cat de probabil e ca acea imagine sa reprezinte template-ul, cu alte cuvinte vrem o masura a similaritatii dintre cele doua. Piesele nu sunt toate simetrice pe ambele axe, dar o piesa de 3 este o piesa de 3 indiferent daca punctele de pe ea sunt orientate de a lungul diagonalei principale sau secundare Astfel, vom roti template-ul in toate cele 4 moduri posibile si calculam pentru fiecare rotatie cat de bine se potriveste cu imaginea. La final, valoarea de potrivire dintre imagine si template este valoarea maxima de potrivire a rotatiilor Pentru a putea utiliza template matching trebuie sa respectam insa constrangerile functiei: template-ul trebuie sa fie mai mic sau egal ca dimeniuni decat imaginea pe care este aplicat, deci, din motive de compatibilitate, redimensionam fiecare rotatie a template-ului la dimeniunile imaginii, adica piesei de domino delimitate. Deoarece template ul are aceleasi dimeniuni cu imaginea pe care se aplica, rezultatul match-ului, in mod normal o imagine de intensitati, este o imagine degenarata intr-un singur punct, punct in care se afla valoarea match-ului, adica gradul de potrivire dintre imagine si rotatia template-ului

```
def template_matching(img, template):
    matches = []
    rotated_template = template
    for _ in range(4):
        rotated_template = cv.rotate(rotated_template,
cv.ROTATE_90_CLOCKWISE)

        rotated_template = cv.resize(rotated_template, (img.shape[1],
img.shape[0]))
        rotated_template = cv.threshold(rotated_template, 127, 255,
cv.THRESH_BINARY)[1]

        res = cv.matchTemplate(img, rotated_template,
cv.TM_CCOEFF_NORMED)

        res_value = res[0][0]

        matches.append(res_value)

    return max(matches)
```

Acum ca avem o functie care ne zice cat de probabil e ca o piesa de domino sa reprezinte un template dat, ne dorim o functie care, dintre toate piesele posibile de zar, sa ne returneze care dintre ele este cea mai probabila sa fie reprezentata pe fata curenta a domino-ului. Pentru aceasta, am creat un folder nou numit **templates** cu imagini decupate **manual** din tabla cu domino-uri pentru fiecare dintre cele 7 fete de zar posibile pe care le vom folosi pentru a testa potrivirea dintre ele si piesa noastra de domino. Cele 7 imagini din folder sunt denumite sub formatul k.PNG, unde k este numarul asociat piesei de zar

Pentru fiecare template din cele 7 il facem alb-negru (grayscale), aplicam un prag ca sa binarizam imaginea (doar valori de alb si negru) ca sa avem aceleasi conditii de prelucrare ca la piesa de domino pe care testam (si ea are doar valori de alb si negru).

Cu ajutorul functiei template_matching definita mai sus, pentru fiecare template calculam potrivirea sa cu fata de zar curenta si retinem aceasdta informatie intr-un dictionar cu perechi de forma (numar_zar - scor_potrivire)

La final, piesa de zar cea mai probabila sa fie reprezentata pe fata curenta a dominoului este cheia din dictionar cu valoarea maxima, drept care asta returnam, alaturi de gradul de incredere al algoritmului in aceasta piesa. De mentionat este ca, din motive de implementare ulterioare, o imagine alba care reprezinta piesa 0 are rezultatul 0, deoarece nu are contrast intre pixeli, drept care are gradul de incredere ori 0 daca imaginea este pur alba, ori un numar mic daca are doar cateva pete negre ce nu reprezinta puncte, asa ca daca gradul de incredere e mai mic decat un prag ales experimental ca fiind 0.07 atunci stabilim ca piesa reprezinta un 0, cu grad de incredere 0 (este rezultatul interpretarii mele experimentale, nu a pattern matching-ului in adevartul sau sens)

```
def identify_domino_piece(source):
    template_matches_values_dict = {}
    for i in range(7):
        template = cv.imread(f"templates/{i}.PNG")

        template = cv.cvtColor(template, cv.COLOR_BGR2GRAY)

        template = cv.threshold(template, 127, 255, cv.THRESH_BINARY)
[1]

        match_value = template_matching(source, template)
        template_matches_values_dict[i] = match_value

    max_value = max(template_matches_values_dict.values())
    if max_value > 0.07:
        return max(template_matches_values_dict.items(), key=lambda x:
x[1])
    else:
        return 0, 0
```

Cum Identificam cate puncte erau pe patch-ul unde s-a pus o piesa? Folosim dictionarul determinat la sectiunea date constante pentru a obtine valoarea asociata cheii date de coordonatele din matricea de joc. Definim o functie care se ocupa de aceasta mapare

Calculul scorului Odata ce avem la fiecare pas scorul pe care il determina fiecare piesa de domino, putem implenenta logica de punctare fara probleme, avand astfel la fiecare pas o evidenta a scorului fiecarui jucator. Ca sa nu avem probleme cu acordarea punctelor(ca se dubleaza dar doar o data si ca daca sunt duble dar se pun pe tabla si are un jucator acolo creste dar o data nu de doua ori etc) retinem cresterea scorului pentru fiecare jucator si facem verificarile de traseu de scor pentru bonus 3 puncte doar pe baza pozitiei de la inceputul mutarii si returnam doar cu cat a crescut scorul fiecarui jucator

```
def get_score_impact_of_domino_piece(first_half_position,
second_half_position, first_half_domino_piece,
second_half_domino_piece, current_player, player_1_points,
player_2_points):
    player_1_gain = 0
    player_2_gain = 0
```

```
    score = get_score_from_position(first_half_position) +
get_score_from_position(second_half_position)

    if current_player == 1:
        player_1_gain += score
        if first_half_domino_piece == second_half_domino_piece:
            player_1_gain += score
    else:
        player_2_gain += score
        if first_half_domino_piece == second_half_domino_piece:
            player_2_gain += score

    player_1_dice_number = scores_to_dice_number[player_1_points]
    player_2_dice_number = scores_to_dice_number[player_2_points]

    if player_1_dice_number == first_half_domino_piece or
player_1_dice_number == second_half_domino_piece:
        player_1_gain += 3
    if player_2_dice_number == first_half_domino_piece or
player_2_dice_number == second_half_domino_piece:
        player_2_gain += 3

    return player_1_gain, player_2_gain
```

Organizarea in structura repetitiva Includem toti pasii utili definiti anterior intr-o structura repetitiva. Deoarece am modularizat fiecare actiune relevanta e mai usor acum sa punem cap la cap bucatile de cod. La fiecare pas vom rula acesti algoritmi pe doua imagini: imaginea curenta si imaginea precedenta, la primul pas imaginea precedenta fiind chiar imaginea cu tabla goala, de start. Vom deschide un folder nou pentru rezultate unde vom adauga pentru fiecare imagine un fisier text cu schimbarile efectuate. Obtinem pentru fiecare "meci" pozele cu tabla de joc. Ulterior, pentru a evalua performanta algoritmului pe masura ce am facut modificari de parametri pentru operatiile de deschidere, blurare, threshold-uire, etc precum si pentru a testa corectitudinea unor noi idei incercate (rotirea imaginilor pentru alinierea lor, decuparea piesei curente din imaginea diferenta vs imaginea originala, obtinerea coordonatelor din centrul piesei pe baza coordonatelor vs prin split-uirea imaginii in 225 de patch-uri, translatii, calcul de margini, centarri, expandari, micsorari si altele)

Pentru a trata cazurile in care patch-ul pe care incercam sa il clasificam are dimeniunea mai mica decat cea a unei casute de pe tabla de joc, umplem imaginea cu patch-ul cu valori albe, deoarece experimental am observat ca in general sunt situatii in care o margine alba a domino-ului nu e inclusa complet in patch

Cum identificam coordonatele unei piese de domino? In urma mai multor abordari incercate, aceasta este cea care s-a dovedit a fi cea mai precisa si rapida. **Ideea de baza** Tabla de joc, din moment ce are dimensiunile de 15 x 15 casute, ea este practic impartita in 225 de patratele, patch-uri, de dimeniunei aproximativ egale. Astfel, a identifica in ce zona din tabla de joc a fost amplasata o piesa este echivalent cu a determina care sunt cele doua casute din imaginea diferenta(eventual cu prelucrari de deschideri / inchideri inainte, determinate experimental) cu cele mai mari valori in medie, adica cu cea mai mare schimbare **Solutia** Implementam doua functii ajutatoare:

- **split_image_into_patches** pentru a sparge imaginea diferenta in casutele de pe tabla de joc, retinand pentru fiecare patch de imagine rezultat coordonatele asociate
- **get_two_highest_values** pentru a identifica cele doua patch-uri cu valoarea schimbarii maxima asociate cu coordonatele lor

Din motive de optimizare, functia de split calculeaza patch-urile delimitate tinand cont de un decalaj minor ce provine din diferenta initiala mica dintre lungimea si latimea pentru tabla de joc si asociaza fiecarui patch coordonatele impartind la 126, numar determinat geometric si experimental

```
def split_image_into_patches(image):
    height, width = image.shape
    average_size = (height + width) // 2

    image = cv.resize(image, (average_size, average_size))

    average_patch_size = average_size // 15

    size_difference_height = height / 15 - average_patch_size
    size_difference_width = width / 15 - average_patch_size

    patches = []

    patches = []
    for i_iter, i in enumerate(range(0, average_size -
average_patch_size + 1, average_patch_size)):
        line_deviation = int(size_difference_height * i_iter)
        for j_iter, j in enumerate(range(0, average_size -
average_patch_size + 1, average_patch_size)):
            column_deviation = int(size_difference_width * j_iter)

            line_start_index = int((i + line_deviation) * 1)
            line_end_index = int((i + average_patch_size +
line_deviation) * 1)
            column_start_index = int((j + column_deviation) * 1)
            column_end_index = int((j + average_patch_size +
column_deviation) * 1)

            patch =
image[line_start_index:line_end_index,column_start_index:column_end_in
dex]
            count = cv.countNonZero(patch)
            patches.append((patch, i // 126 + 1, j // 126 + 1, count,
line_start_index, line_end_index, column_start_index,
column_end_index))
    return patches
```

Ajustarea imaginilor Pe baza coordonatelor obtinute din spargerea initiala a imaginii in patch-uri nu se obtine, pentru un patch cu o jumatate de piesa de domino, fix imaginea intreaga cu patch-ul. In ciuda optimizarilor aduse functiei de split-uire a imaginii, tabla de joc are mici

deviatii de la impaturirea / plierea ei (pare ca tabla se tine impaturita in 4 bucati ca un servetel cand e in cutie). Mai mult, piesele nu sunt amplasate drept si fix absolut pe casute, unele sunt puse un pic stramb (au o inclinatie) iar altele, in special cele de pe margini, nu sunt prinse in intregime. In cele ce urmeaza vom aplica metode geometrice, eursitice si bazate pe muchii si gradienti pentru a obtine dintr-un patch translatarea sa cat mai buna astfel incat piesa de domino sa fie cat mai centrala pe patch Ulterior, pentru a imbunatii acuratetea predictiilor, inainte de a aplica template matching-ul pe piesele translatate aplicam anumite operatii, determinate si optimizate experimental, pentru aducerea imaginii cu dominoul la o forma cat masi apropiata de cea a template-urilor pentru a imbunatatii gradul de incredere al predictiilor adevarate si a evita situatiile in care un domino este misclasificat din cazua necentrarii lui sau a zgomotului din imagine

Translatarea patch-ului Ideea principala este urmatoarea: o piesa centrata are zone negre, cu fundalul de la tabla de joc, neglijabile in jurul ei, in timp ce o piesa necentrata are o zone la stanga, dreapta, sus, jos care au predominant valori de negru, formand astfel o margine. **Ce ne dorim?** Vrem pentru o imagine cu un domino sa calculam care sunt marginile negre la stanga, dreapta, sus, jos ca apoi sa translatam piesa, adica sa modificam coordonatele dupa care o cropam din imaginea sursa **Solutii** Avem doua abordari pentru a realiza acest lucru:

- **Euristica**: calculam pentru fiecare latura a patch-ului care este cea mai indepartata linie paralela cu aceasta care are un procentaj de valori de pixeli non-albi mai mare decat un prag, insa aceasta linie trebuie sa fie la o distanta de cel mult o anumita fractiune din dimeniunea imaginii originale de margine
- **Identificarea muchiilor**: pentru un patch dat, identificam muchiile sale pe baza gradientului Canny si apoi folosim Transformata Hough pentru a determina ce linie corespunde fiecarui punct. Dupa ce obtinem coordonatele liniilor din imagine, determinam care sunt verticale si care sunt orizontale si retinem coordonata care le descrie (x pentru liniile verticale, y pentru cele orizontale). Bazandu-ne pe faptul ca muchiile apar unde gradientii sutn mari, adica unde avem pixeli albi langa pixeli non-albi, practic aceste muchii apar la granita piesei efective de domino. Cu cat o linie verticala este mai la dreapta intr-un patch, cu atat inseamna ca patch-ul respectiv e decalat la dreapta, deci putem echivala marginea sa neagra cu coordonata maxima a muchiilor verticale din imagine situate in fractiunea stanga din imagine. Analog se determina si marginile negre din dreapta, sus si jos

Euristica (Heursitic Border - HB) Prin experimente repetate, am ajuns la concluzia ca o valoare de prag optima pentru procentajul de pixeli non-albi dintr-o linie pentru a cosnidera corect ca aceasta face parte din margine este de 0.7, in timp ce fractiunea considerata pe fiecare latura pentru muchiile corespunzatoare este de 1/3.

```
def get_average_black_border_size_enchanced(img):
    black_pixels_percentage_threshold = 0.7
    black_border_left, black_border_right, black_border_top,
black_border_bottom = 0, 0, 0, 0

    for j in range(int(img.shape[1] // 3)):
        total_pixels = img.shape[0]
        black_pixels = total_pixels - cv.countNonZero(img[:, j])
        if black_pixels / total_pixels >
```

```
black_pixels_percentage_threshold:
            black_border_left = j

    for j in range(img.shape[1] - 1, img.shape[1] - int(img.shape[1]
// 3) - 1, -1):
        total_pixels = img.shape[0]
        black_pixels = total_pixels - cv.countNonZero(img[:, j])
        if black_pixels / total_pixels >
black_pixels_percentage_threshold:
            black_border_right = img.shape[1] - j

    for i in range(int(img.shape[0] // 3)):
        total_pixels = img.shape[1]
        black_pixels = total_pixels - cv.countNonZero(img[i, :])
        if black_pixels / total_pixels >
black_pixels_percentage_threshold:
            black_border_top = i

    for i in range(img.shape[0] - 1, img.shape[0] - int(img.shape[0]
// 3) -1, -1):
        total_pixels = img.shape[1]
        black_pixels = total_pixels - cv.countNonZero(img[i, :])
        if black_pixels / total_pixels >
black_pixels_percentage_threshold:
            black_border_bottom = img.shape[0] - i

    return black_border_left, black_border_right, black_border_top,
black_border_bottom
```

Muchii (Canny apoi Hough - CH) Convertim imaginea la alb-negru daca nu este deja, apoi inversam culorile de lab si negru aplicand operatia de negare pe biti pentru a obtine un contrast mai bun, deci muchii mai bune. Obtinem conturul, gradientul prin metoda Canny si apoi aplicam transformata Hough pe imaginea rezultat pentru a obtine liniile pe care le clasificam ca fiind verticale sau orizontale si retinem coordonatele asociate. Pentru fiecare set de coordonate egalam marginea corespunzatoare cu valoarea maxima sau 0 daca setul este gol(daca este gol inseamna ca nu exista muchii pe directia respectiva deci piesa nu are margine pe latura corespunzatoare)

```
def identify_black_lines(image, threshold=50, line_length=100,
line_gap=10):
    if len(image.shape) > 2:
        gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    else:
        gray = image

    inverted = cv.bitwise_not(gray)

    edges = cv.Canny(inverted, 50, 150, apertureSize=3)
```

```
    lines = cv.HoughLinesP(edges, 1, np.pi/180, threshold,
minLineLength=line_length, maxLineGap=line_gap)

    line_image = np.zeros_like(image)

    vertical_lines_x_coordinates = []
    horizontal_lines_y_coordinates = []

    if lines is not None:
        for line in lines:
            x1, y1, x2, y2 = line[0]
            cv.line(line_image, (x1, y1), (x2, y2), (255, 0, 0), 2)

            if x1 == x2:
                vertical_lines_x_coordinates.append(x1)
            elif y1 == y2:
                horizontal_lines_y_coordinates.append(y1)

    left_border, right_border, top_border, bottom_border = 0, 0, 0, 0

    left_values = [x for x in vertical_lines_x_coordinates if x <
image.shape[1] // 2]
    if len(left_values) != 0:
        left_border = max(left_values)

    right_values = [x for x in vertical_lines_x_coordinates if x >
image.shape[1] // 2]
    if len(right_values) != 0:
        right_border = image.shape[1] - min(right_values)

    top_values = [y for y in horizontal_lines_y_coordinates if y <
image.shape[0] // 2]
    if len(top_values) != 0:
        top_border = max(top_values)

    bottom_values = [y for y in horizontal_lines_y_coordinates if y >
image.shape[0] // 2]
    if len(bottom_values) != 0:
        bottom_border = image.shape[0] - min(bottom_values)

    return line_image, left_border, right_border, top_border,
bottom_border
```

Aplicarea translatiilor Dupa ce am identificat prin una dintre cele doua metode descrise mai sus marginile negre, translatam imaginea pe ambele axe. **Exemplu**: Daca o piesa are o margine neagra la stanga de 20 de pixeli si la dreapta de 2 pixeli, imaginea trebuie translatata la dreapta cu diferenta dintre cele doua, adica cu 18 pixeli. Daca am translata doar pe baza unei margini putem risca sa decalam piesa prea mult in cealalta parte. Astfel obtinem translatia orizontala si, in mod analog, si translatia verticala. Aceste translatii se aplica pe coordonatele de start si final

ale unui patch dintr-o imagine sursa, asa ca trebuie sa verificam ca noii indecsi de delimitare sa nu fie in afara imaginii sursa, caz in care nu aplicam translatia pe indexul respectiv

Pentru modularizare si experimentare ulterioara, wrap-uim cei doi algoritmi de calculare a marginilor si translatarile ulterioare in doua functii

Deoarece vom aplica acesti algoritmi succesiv (explicat de ce mai incolo), definim si o functie care realizeaza operatia lor comuna de identificare a marginilor si translatiilor la fiecare pas, actualizand patch-ul prin coordonatele translatate. Aceasta functie primeste in plus instructioni, adica o lista de siruri de caractere care denota operatiile de translatare cu algoritmul euristic (Heurisitc Borders, deci "HB") sau cu cel bazat pe muchii (Canny apoi Hough, deci "CH")

```
def get_translated_patch(patch, patch_line_start_index,
patch_line_end_index, patch_column_start_index,
patch_column_end_index, source_image, instructions):
    for instruction in instructions:
        if instruction == "CH":
            patch, patch_line_start_index, patch_line_end_index,
patch_column_start_index, patch_column_end_index =
apply_canny_hough_lines_translation(patch, source_image,
patch_line_start_index, patch_line_end_index,
patch_column_start_index, patch_column_end_index)
        elif instruction == "HB":
            patch, patch_line_start_index, patch_line_end_index,
patch_column_start_index, patch_column_end_index =
apply_heuristic_black_borders_translation(patch, source_image,
patch_line_start_index, patch_line_end_index,
patch_column_start_index, patch_column_end_index)
        else:
            continue

    return patch
```

Preprocesarea piesei de domino Dupa ce am aplicat translatiile si am obtinut un patch mai centrat, inainte de a trece la etapa de identificare cu template matching, aplicam anumite operatii peste piesa de domino pentru a elimina zgomotul pana ramanem cu o imagine care contine doar un fundal alb cu forme cat de cat sferice pentru punctele de pe piesa de domino Pentru acest lucru aplicam urmatoare operatii, in aceasta ordine:

- Expandare
- Dilatare
- Inchidere
- Filtrare cu filtru median
- Completare cu alb
- Inchidere
- Albirea marginii
- Centrarea imaginii
- Taiere

Expandare Marim dimensiunile imaginii cu un procentaj din cele initiale in mod simetric si umplem aceasta regiune cu valori albe

Centrare Obtinem din imagine centrul continutului sau ca fiind media coordonatelor unde se afla valori de negru (adica cercurile de pe zar), calculam diferenta dintre centrul imaginii si centrul continutului imaginii pentru a obtine translatiile orizontale si verticale pentru imagine si apoi aplicam aceste translatii

```
def middle_point(image):
    total_x = 0
    total_y = 0
    count = 0

    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            if image[y, x] == 0:
                total_x += x
                total_y += y
                count += 1

    if count == 0:
        return None

    average_x = total_x / count
    average_y = total_y / count

    return average_x, average_y

def center_image(image, center, desired_center):
    if center is not None:
        horizontal_translation = int(desired_center[0] - center[0])
        vertical_translation = int(desired_center[1] - center[1])

        image = np.roll(image, horizontal_translation, axis=1)
        image = np.roll(image, vertical_translation, axis=0)

    return image
```

Taiere Pastram doar un patch (din patch-ul initial) de dimeniuni proportionale cu cele initiale si simetric (decupam in mod egal de pe fiecare latura)

```
def apply_patch_filling(patch, desired_size, patch_line_start_index,
patch_line_end_index, patch_column_start_index,
patch_column_end_index, source_image, instructions):
    patch = get_translated_patch(patch, patch_line_start_index,
patch_line_end_index, patch_column_start_index,
patch_column_end_index, source_image, instructions)

    patch = expand_image(patch, 0.2)
```

```
    kernel1 = np.ones((10, 10), np.uint8)
    patch = cv.dilate(patch, kernel1, iterations=1)

    patch = get_closed_image(patch, 4)

    patch = cv.medianBlur(patch, 5)

    patch = complete_with_white(patch, desired_size)

    patch = get_closed_image(patch, 12)

    patch[:, int(patch.shape[1] * 0.95):] = 255
    patch[:, :int(patch.shape[1] * 0.05)] = 255
    patch[:int(patch.shape[0] * 0.05), :] = 255
    patch[int(patch.shape[0] * 0.95):, :] = 255

    center_of_points = middle_point(patch)
    desired_center = (patch.shape[1] // 2, patch.shape[0] // 2)

    patch = center_image(patch, center_of_points, desired_center)

    patch = cut_border(patch, 0.2)

    return patch
```

De ce avem nevoie de doua moduri de a identifica marginile negre? Experimental am observat ca cea mai dificila sarcina este cea de clasificare a domino-ului si ca principalul factor care conducea la misclaisificari era faptul ca piesele de domino decupate din imaginea originala nu erau mereu incadrate perfect de coordonatele asociate prin spargerea imaginii in 225 de patch-uri, ramanand acele margini negre. Chiar daca cei doi algoritmi rezolva acelasi task, o fac in moduri usor diferite si experimental am observat ca existau cazuri in care daca aplicam unul dintre ei mergea bine pe anumite cazuri dar mai putin bien pe alte cazuri, in timp ce daca il aplicam pe celalat era viceversa. Mai mult, am observat si ca daca aplicam acesti algoritmi unul dupa altul rezultatele difera. Prin aplicarea unei singure combinatii de algoritmi se obtine undeva la 95-98% din scorul total. Tot experimental am observat ca cele mai bune combinatii de algotitmi sunt, nu neaparat in aceasta ordine: HB + CH, CH + HB, CH si HB Mai mult, pe cazurile simple, evidente, cu imagini bine centrate din prima, ori toti algoritmii ori o majoritate din ei (3 din 4) conduc la acelasi rezultat final al clasificarii piesei de domino. In cazul in care in urma unui algoritm o piesa este clasificata gresit, acest lucru se intampla, observat experimental, din cauza delimitarii defectuase si piesa finala nu seamana prea mult cu template-urile, drept care gradul de incredere in predictie este relativ scazut. Pe de alta parte, un algoritm care conduce la o delimitare mai buna a piesei rezulta intr-o predictie cu un grad mai mare de incredere, deci care s-a potrivit destul de mult cu unul dintre template-uri. Astfel, putem spune ca, din moment ce avem un algoritm "mai sigur pe el", alegem rezultatul provenit in urma aplicarii lui ca fiind adevarata clasificare **Solutia?** Obtinem clasificarea piesei de domino in urma aplicarii fiecaruia dintre cei 4 algoritmi si retinem pentru fiecare valoarea determinata si gradul de incredere in predictia facuta. Daca exista o valoare care a fost determinata de un numar maxim de ori in raport cu celelalte, atunci stabilim ca valoarea respectiva este cea adevarata ca urmare a votului majoritatii. Altfel, daca exista mai multe valori care apar de acelasi numar maxim de ori (exemplu 3 de doua ori, 5 de doua ori), pentru fiecare dintre cele doua valori calculam suma gradelor de

incredere asociate predictiilor si alegem valoarea piesei de domino ca fiind valoarea care a generat suma maxima a gradelor de incredere

```
def get_vote_results(votes):
    max_count = max([len(votes[key]) for key in votes.keys()])

    if max_count == 0:
        return None

    max_keys = [key for key in votes.keys() if len(votes[key]) ==
max_count]

    if len(max_keys) == 1:
        return max_keys[0]

    else:
        max_sum = 0
        max_key = None
        for key in max_keys:
            current_sum = sum(votes[key])
            if current_sum > max_sum:
                max_sum = current_sum
                max_key = key

        return max_key
```

Implementarea finala Inglobam functionalitatile definite anterior intr-o bucla repetitiva pentru fiecare meci cu fiecare imagine din meci. Pentru a cuantifica rezultatele modificarilor experimentale aduse. Practic, la fiecare pas, identificam regiunea unde s-a adaugat o piesa, obtinem doua patch-uri corespunzatoare celor doua valori de pe piesa de domino si actualizam scorul pe baza valorilor determinate

Concluzii Solutia propusa este rezultatul a zile intregi de experimentat cu tehnici si parametri. Ideea initiala de a rezolva problema difera aproape complet de rezolvarea actuala intrucat am adus mici modificari pe parcurs pana cand algoritmul s-a schimbat aproape complet. A fost un proiect din care am invatat multe si la care chiar mi-a facut placere sa lucrez, fiind cel mai complex si muncit proiect pe care l-am facut pana acum de cand sunt student.

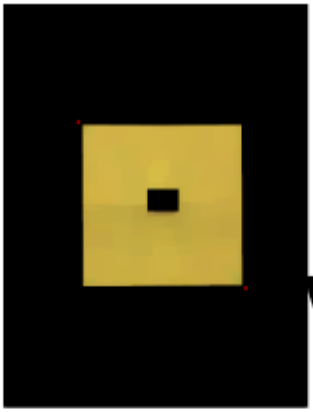
masca

open

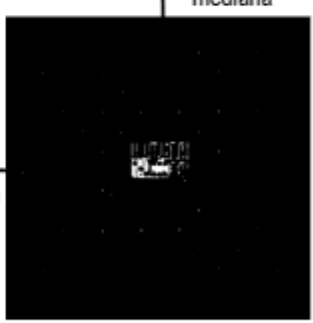
procesarea

diferentei

coordonatele tablei de joc



blurare
mediana



top 2

patches

analog 3

predictie 1

analog

algoritm(i)
translatie

original

image patch

expandare

dilatare

predictie 2