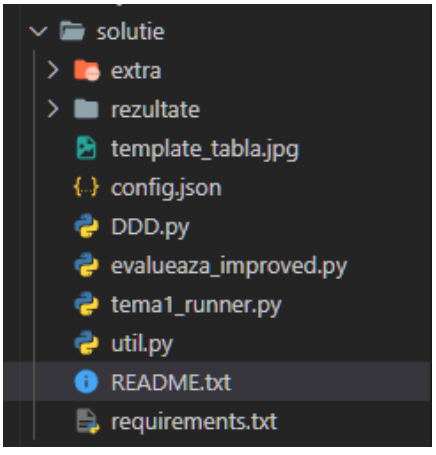
Licu Mihai-George CTI 462

Descrierea Directorului

1. **Directorul Extra:**
   - **Funcționalitate:** Acest director conține o serie de scripturi ajutătoare și notebook-uri Jupyter utilizate pentru modularizarea procesului de rezolvare a diferitelor sarcini și construirea de pipeline-uri.
   - **Rol:** Deși nu este esențial pentru rularea proiectului, directorul 'extra' este inclus ca dovadă a conceptului și a procesului de dezvoltare.
2. **Directorul Rezultate:**
   - **Funcționalitate:** Acest director este inițial gol, facilitând rularea runner-ului cu parametrii default fără erori.
   - **Rol:** Servește ca spațiu de stocare pentru rezultatele generate în timpul execuției.
3. **Fișierul template_tabla.jpg:**
   - **Importanță:** Este esențial pentru inițierea fiecărui joc.
   - **Instrucțiuni:** Dacă acest fișier lipsește, este necesar să se specifice în config.json un path către o imagine a tablei de joc goale.
4. **Fișierul DDD.py:**
   - **Funcționalitate:** Reprezintă clasele pentru abstractizarea jocului, efectuarea game loop-ului și conține logica principală a scorului.
5. **Fișierul evalueaza_improved.py:**
   - **Descriere:** Este un evaluator îmbunătățit, cu culori și comparații superioare.
6. **Fișierul tema1_runner.py:**
   - **Rol:** Funcționează ca runner pentru fiecare task al proiectului.
7. **Fișierul util.py:**
   - **Conținut:** Prezintă toate funcțiile ajutătoare dezvoltate pentru compunerea diverselor pipeline-uri.
8. **Fișierele README și requirements:**
   - **README:** Conține instrucțiuni de utilizare și descrierea proiectului.
   - **requirements:** Enumeră bibliotecile necesare pentru rularea proiectului.



Licu Mihai-George CTI 462

Abordarea Generală

Această secțiune descrie abordarea adoptată în gestionarea procesului de joc pentru proiect. În loc de a utiliza procesarea imaginilor pentru a urmări scorul sau punctele bonus, proiectul se bazează pe o metodă mai directă și eficientă.

Motivația Deciziei

- **Scorul și punctele bonus:** Layout-ul lor este constant de la un joc la altul, nu necesită determinarea prin procesarea imaginilor.
- **Reprezentarea internă a scorului:** Bazată pe mutări, acesta oferă o abordare simplificată și mai controlabilă.

Implementarea Procesului de Joc

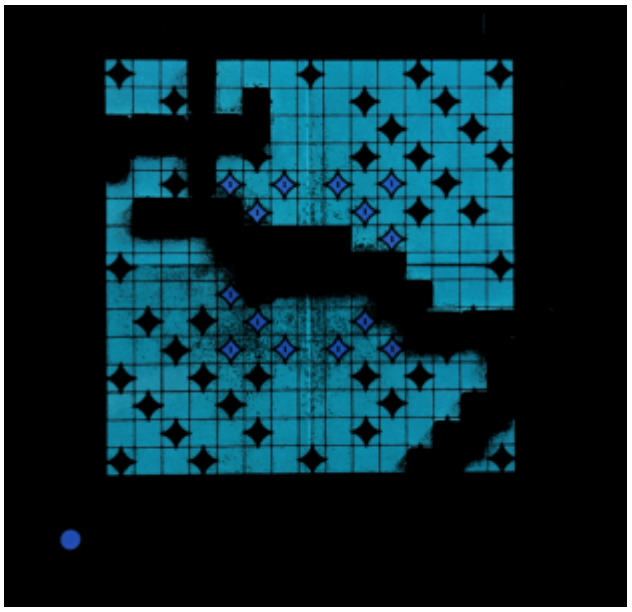
1. **Inițializarea Obiectelor de Joc:**
   - Fiecare joc DDD este reprezentat printr-un obiect Game.
   - Inițializarea include: ID-ul jocului, fișierul cu mutări și jucătorul care le efectuează, imaginea inițială a stării jocului (inițial tabla goală), descrierea tablei și a trackerului de scor (citite din fișierul de configurație), un dicționar pentru scorul fiecărui jucător, și fișierele de intrare/ieșire pentru imagini și predicții.
2. **Abstractizarea Logică a Game Loop-ului:**
   - **Extracția Grid-ului:** Din prima imagine se extrag liniile grid-ului, care rămân relativ constante pe parcursul jocului. Se păstrează grid-ul pentru optimizare, dar există posibilitatea actualizării acestuia dacă perspectiva imaginii se schimbă.
   - **Iterarea prin Mutări:** Procesul parcurge 20 de mutări, executând următorii pași pentru fiecare:
     - Citirea imaginii noi și a jucătorului care a efectuat mutarea.
     - Cropping la regiunea de interes pentru a detecta noua piesă plasată.
     - Actualizarea stării scorului.
     - Afișarea mutării și actualizarea prev_image cu noua imagine.

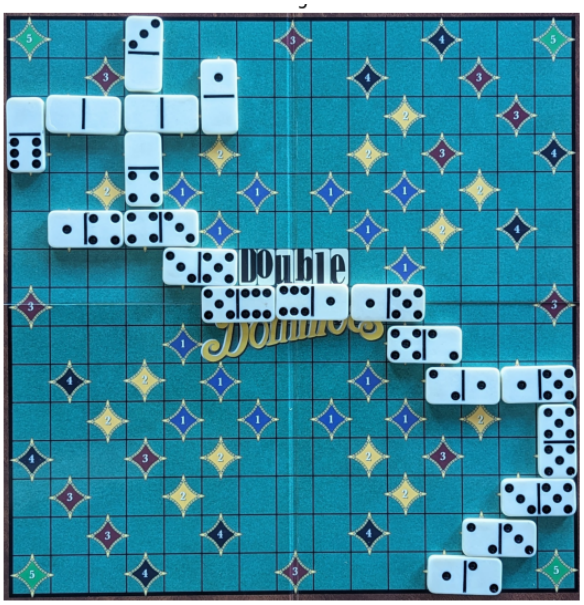
```python
def loop(self):
    self.grid = self.get_grid(self.prev_image.copy())
    for move_num in range(1, 21):
        jpg_path, player = self.mutari[move_num - 1].split()

        image_path = self.input_dir + jpg_path
        self.cur_image = crop_board(cv.imread(image_path))
        piece = self.detect_piece()
        points = self.place(piece, player, move_num)
        self.print_move_info(piece, points, move_num)
        self.prev_image = self.cur_image.copy()
```

Licu Mihai-George CTI 462

Procesul de Crop-uire și Detectare a Grid-ului

1. **Crop-uirea Board-ului:**
   - **Metodă:** Am observat ca dreptunghiul intern care este principala regiune de interes este alcatuita in principal din aceeași culoare unică în imagine, așadar am ales sa aplic o masca cu un filtru de culoare folosind valorile HSV (hMin: 95, sMin: 130, vMin: 135, hMax: 140, sMax: 255, vMax: 255) pentru a izola regiunea principală de interes.
   - **Identificarea Conturului:** Se găsește cel mai mare contur și se construiește un dreptunghi în jurul acestuia. Imaginea inițială este apoi crop-uită la dimensiunile dreptunghiului și se schimbă perspectiva.





   -

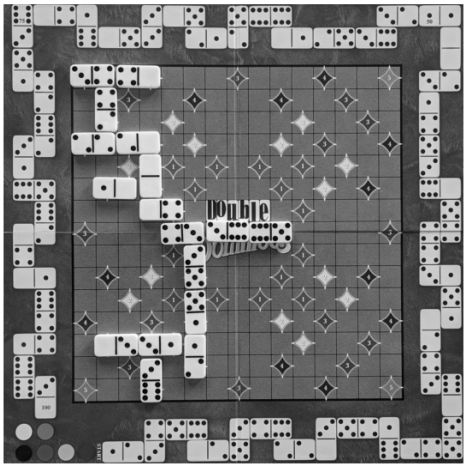
2. **Descoperirea Grid-ului:**
   - **Pipeline de Procesare:** Include denoise, sharpening, edge detection.
   - **Identificarea Liniilor:** Se găsesc liniile relativ orizontale sau verticale, se clusterizează liniile apropiate pentru a păstra 32 de linii (16 orizontale și 16 verticale).
   - **Utilizarea Rho și Theta:** Se reține valorile rho și theta ale liniilor pentru utilizare ulterioară.

O prima idee de am avut-o pentru pentru denoise si sharpening a fost folosirea unei măști in forma de cruce (+) gandindu-ma ca acest lucru ar păstra cel mai bine liniile, dar după diverse încercări am observat ca un simplu filtru bilateral pastreaza cel mai bine liniile si o operație de sharpening cu doua kerneluri (unul orizontal si unul vertical) dau cele mai bune rezultate. Apoi doar aplic Canny edge detection pe imagine.

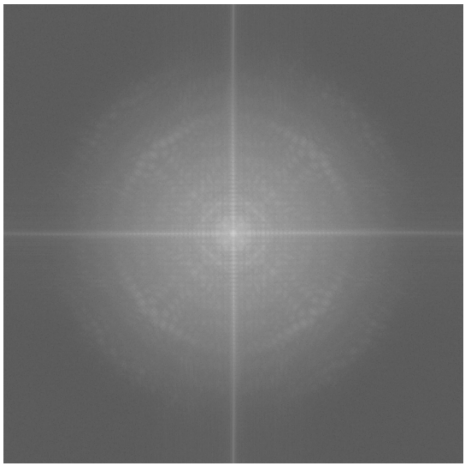
Licu Mihai-George CTI 462

Gaussian

Image

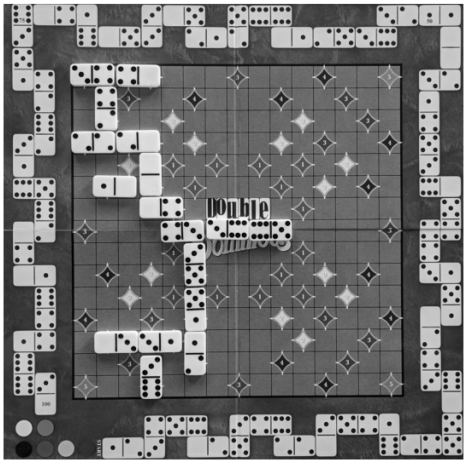


Spectrum

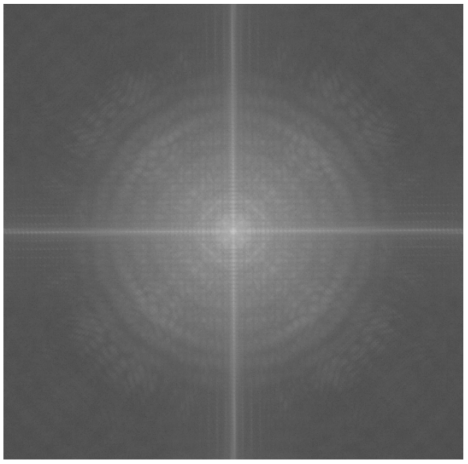


Bilateral

Image

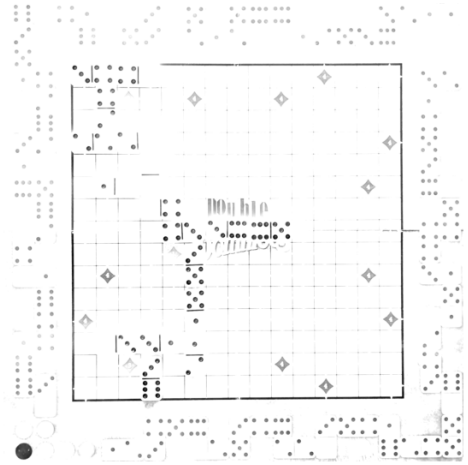


Spectrum

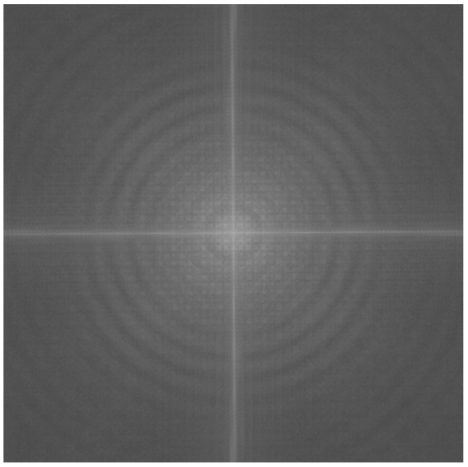


Sharpened

Image

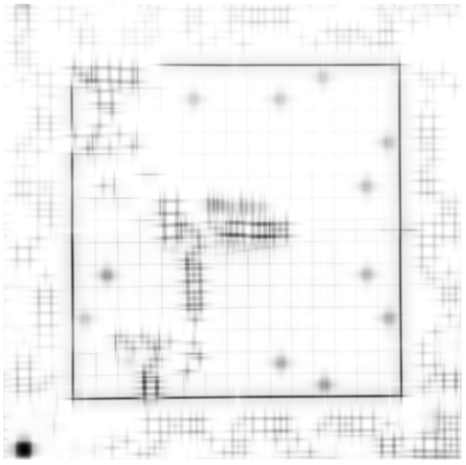


Spectrum

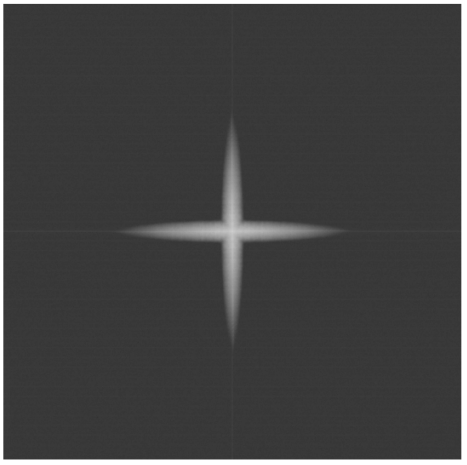


Masked

Image



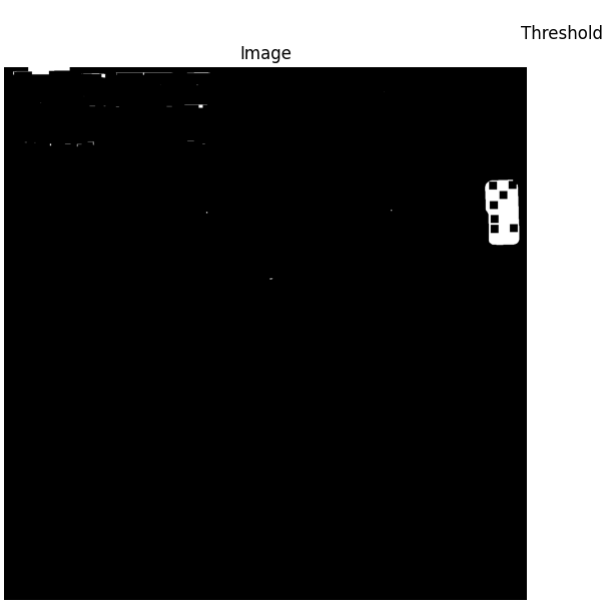
Spectrum



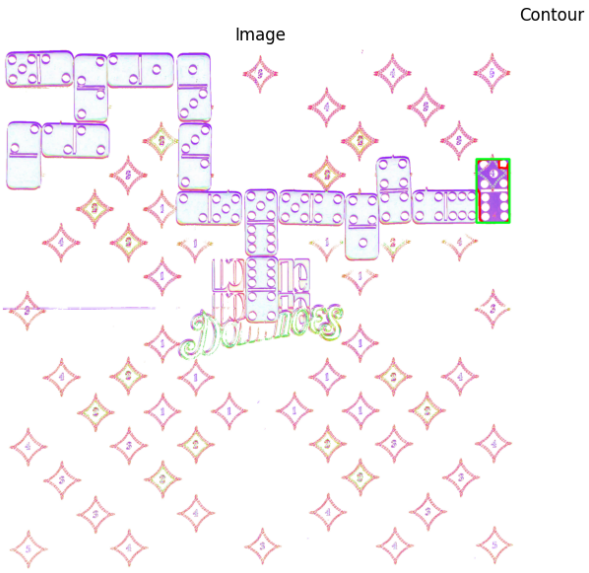
Licu Mihai-George CTI 462

Detectarea Piesei Plasate

1. **Preprocesarea Imaginilor:**
   - **Filtru RGB:** Similar cu cel anterior, se păstrează doar zonele de culoare asemănătoare cu piesele de joc.
   - **Redimensionare și Diferență:** Imaginile sunt redimensionate la cea mai mică valoare comună și se calculează diferența absolută pentru a obține o nouă mască.
2. **Identificarea și Procesarea Piesei:**
   - **Filtrare și Eroziune:** Se aplică un threshold binar și o eroziune mică, urmată de o operație de closing pentru a elimina artefactele.
   - **Detectarea Conturului:** Se găsește cel mai mare contur cu un aspect ratio așteptat pentru un domino (2:1), cu o marja de eroare.
   - **Procesarea Dominoului:** Se împarte aria dominoului în două și se utilizează HoughCircles pentru a identifica cercurile ce indică valoarea piesei (cv.HOUGH_GRADIENT_ALT, dp=1, minDist=25, param1=400, param2=0.2, minRadius=8, maxRadius=16).

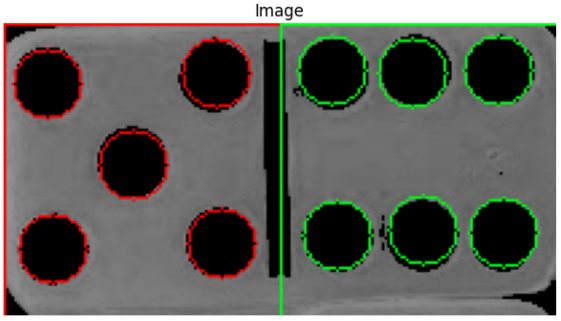


Threshold



Contour

Domino



Licu Mihai-George CTI 462

Algoritmul de Plasare pe Tablă

1. **Determinarea Poziției pe Tablă:**
   - Se calculează aria dreptunghiului piesei între fiecare două linii ale grid-ului, identificându-se cele două pătrate unde dreptunghiul are cea mai mare arie.
2. **Actualizarea Stării Jocului:**
   - Se construiește un nou obiect Piece cu două Squares.
   - Se actualizează starea scorului.

```
def localize_piece(
    rect: Tuple[float, float, float, float], lines: Dict[str, List[Tuple[float, float]]]
) -> List[Tuple[int, int]]:
    x, y, w, h = rect
    vertical_lines = sorted([line[0] * np.cos(line[1]) for line in lines["vertical"]])
    horizontal_lines = sorted(
        [line[0] * np.sin(line[1]) for line in lines["horizontal"]]
    )
    def find_overlapping_squares():
        overlapping_squares = []
        for i in range(len(horizontal_lines) - 1):
            for j in range(len(vertical_lines) - 1):
                if (
                    horizontal_lines[i] < y + h
                    and horizontal_lines[i + 1] > y
                    and vertical_lines[j] < x + w
                    and vertical_lines[j + 1] > x
                ):
                    overlapping_squares.append((i, j))
        return overlapping_squares

    overlapping_squares = find_overlapping_squares()
    intersection_areas = {}
    for i, j in overlapping_squares:
        top = max(y, horizontal_lines[i])
        bottom = min(y + h, horizontal_lines[i + 1])
        left = max(x, vertical_lines[j])
        right = min(x + w, vertical_lines[j + 1])

        area = max(0, right - left) * max(0, bottom - top)
        if area > 0:
            intersection_areas[(i, j)] = area

    top_two_squares = sorted(
        intersection_areas, key=intersection_areas.get, reverse=True
    )[:2]

    top_two_squares = sorted(top_two_squares, key=lambda x: (x[1], x[0]))
    return top_two_squares
```