CV Final Project Report

No	Name	NetID
1	Tanmay Rathi	tr2452
2	Simardeep Singh Mehta	sm11377
3	Avadhoot Kulkarni	ak10576

Automated Chess Board Recognition and Interaction using a top-view setup

Abstract

This project presents a comprehensive system for the recognition and interaction with a chessboard and its pieces using a top-view imaging approach. By simplifying the visual input to a two-dimensional plane, our solution effectively utilizes image processing techniques to detect and classify chess pieces, ensuring an accurate representation of the game state which helps in further automated analysis or interactive gameplay.

Project Motivation

Throughout our course, we've explored various image processing techniques, such as edge detection, Hough transforms, filtering, and template matching. This project offered us a unique opportunity to apply these techniques to a real-world problem—automating the detection and tracking of chess piece movements in real-time.

The game of chess involves complex strategic and tactical play, where the ability to accurately recognize the position and type of pieces on a chessboard is crucial. Traditional methods, involving multi-angle views and 3D reconstruction, often complicate the direct application of image processing techniques due to the variability in piece appearance from different angles. Our project addresses this challenge by implementing a top-view setup that standardizes the visual input and simplifies the processing requirements.

In our implementation, we capture video frames at key moments—specifically, when a player makes a move and hits the clock. By comparing the edge magnitude images of consecutive frames, we identify movement peaks. Adjustments such as cleaning up noise and zeroing out pixels along the detected grid lines of the chessboard help refine the detection process. This method has proven not only efficient, with processing times under one second, but also nearly real-time, making it a viable solution for automated chess systems in competitive and casual environments alike.

Automated chess tracking presents a viable alternative to the current standards, which often rely on high-cost sensor-equipped boards or software that depends heavily on deep learning or template matching. While deep learning offers powerful analytical capabilities, it is resource-intensive, requiring substantial computational power and extensive training datasets. Template matching, on the other hand, although simpler, lacks flexibility and is highly sensitive to changes in piece design, orientation, and scale. This method also faces performance issues, as it can be computationally demanding and slow, particularly when matching each chess piece across numerous possible positions. Our approach leverages a stable camera setup and straightforward computer vision techniques. It is cost-effective, minimizing the need for expensive hardware, and efficiently processes frames to enable near real-time tracking. Additionally, it is scalable and can be easily adapted to different environments without extensive reconfiguration, addressing many of the limitations of current technologies.

Introduction:

Edge detection

Edge detection is a technique in image processing used to identify boundaries within an image. It typically involves applying a convolutional operator, such as the Sobel, Prewitt operators, to highlight areas of rapid intensity change. These operators compute gradients in multiple directions, emphasizing regions where intensity changes sharply, indicative of edges. After obtaining gradient magnitudes and orientations, non-maximum suppression is often performed to thin down detected edges to single-pixel widths. Finally, edges can be further refined using techniques like thresholding to retain only strong edges while suppressing weak ones. Edge detection is a preliminary step for the Hough Transform algorithm to detect straight lines in the images. One of the most commonly used methods is the Canny edge detector, accessible via the cv2.Canny() function. This method combines gradient-based edge detection with two thresholds: an upper threshold and a lower threshold to produce high-quality edge maps.

Hough Transform for straight lines

The Hough Transform is a technique used primarily in image processing for detecting shapes, particularly lines or curves, within an image. It works by transforming the Cartesian coordinate space of the image into a parameter space, typically representing lines or curves. In the case of detecting lines, each point in the Cartesian space is mapped to a sinusoidal curve in the Hough space. By accumulating these curves, the intersection points in the parameter space correspond to lines in the original image. The peaks in the accumulator space indicate the presence of lines, allowing for robust detection even in the presence of noise or partial occlusion.

Algorithm for Hough Transform:

Initialize accumulator array A to zeros.
For each feature point (x,y) in the edge-detected image:
    For theta = 0 to 180:
        p =xcos(theta) + ysin(theta)
        A(theta,p) = A(theta,p) + 1
Find local maxima (theta,p) in the accumulator array A.
Extract lines corresponding to the local maxima.
For each local maximum (theta, p) in A:
    Find the corresponding line: p = xcos(theta) + ysin(theta)

Contours

Contours are outlines or boundaries of objects in an image, typically representing changes in intensity or color. They are derived from edge-detected images but often involve additional processing steps to identify and trace the boundaries of connected components. The contour detection process involves identifying connected components of similar intensity or color and tracing their boundaries. Various algorithms, such as the Canny edge detector or the Sobel operator, are commonly used to extract contours from images by detecting abrupt changes in intensity or gradients. Once detected, contours can be further analyzed and utilized for different purposes, such as object recognition, tracking, or measurement.

Saddle Points

Saddle points refer to points in the image where the intensity changes in one direction while it changes in the opposite direction in another direction. Mathematically, these points correspond to locations where both the second-order partial derivatives in different directions have opposite signs. Thus, a saddle point in an image indicates a region where the intensity changes in one direction (e.g., increasing) while simultaneously changing in the opposite direction (e.g., decreasing) when moving along

a different direction (e.g., perpendicular). The strength of the saddle point at each pixel represents the magnitude of this opposing intensity change, which can be calculated using second derivatives.

Douglas-Peucker Algorithm
The Douglas-Peucker algorithm is a well-known technique for reducing the number of points in a curve that is approximated by a series of lines. The goal is to minimize the detail of the contour while maintaining the overall shape, which is crucial for subsequent processing stages like feature detection and object classification.

Morphological Transformations
Morphological transformations are some simple operations based on the image shape. It is normally performed on binary images. It needs two inputs, one is our original image, and the second one is called a structuring element or kernel which decides the nature of operation.

Methodology:

Image Gradient Computation

The first step is to detect the edges and features within our chessboard recognition system in order to simplify subsequent image processing tasks like piece detection and move tracking. We begin by computing the gradients in both the horizontal (x) and vertical (y) directions using the Sobel operator, which is part of the OpenCV library (`cv2.Sobel`). This operator helps highlight areas of significant intensity change, which are indicative of edges.

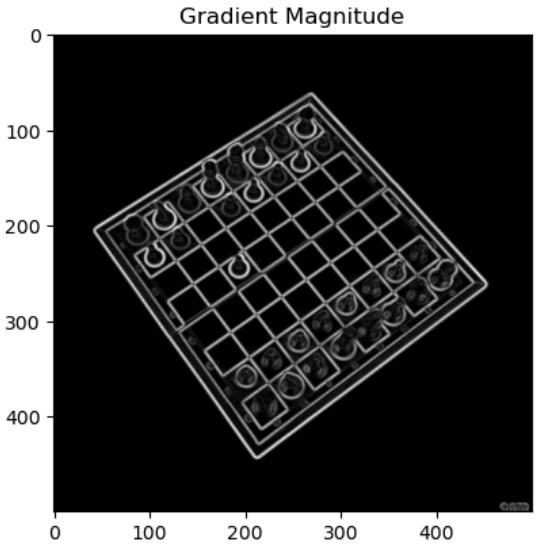
We then proceed to calculate the gradient magnitude and phase. The magnitude of the gradient at each pixel is calculated as the Euclidean norm of the x and y gradients (square root of the sum of the squares of the individual gradients). This step combines the horizontal and vertical changes at each point to produce a single measure of edge strength. The phase (or orientation) of the gradient is calculated using the arctangent of the ratio of the y-gradient to the x-gradient (`np.arctan2`). This indicates the direction of the steepest increase in intensity and is crucial for understanding the orientation of edges.

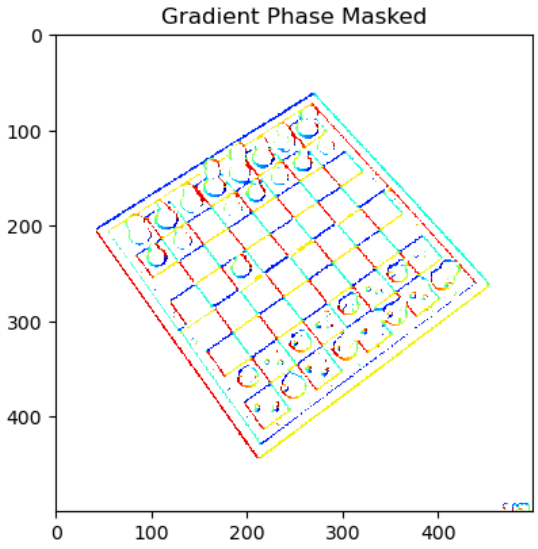
To focus only on significant edges while minimizing noise, we employ thresholding techniques. Gradient magnitude thresholding involves applying a threshold to the

magnitudes, set at twice the average magnitude across the image. This step helps in isolating prominent edges. Additionally, phase masking is performed to mask phases where the gradient magnitude falls below the threshold, effectively excluding phases associated with insignificant intensity changes from further processing.

Input Image

Processed Image

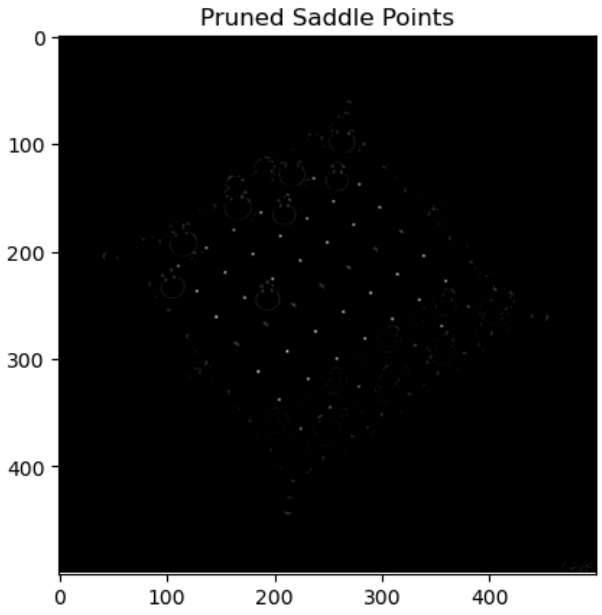




Saddle Point Detection

The next step is to calculate saddle points, which is particularly useful for detecting specific features on a chessboard from a top-view imaging setup. We ensure that the input is a single-channel grayscale image as subsequent operations rely on the image being in the correct format. We begin by calculating the first-order derivatives gx and gy in the horizontal and vertical directions respectively using the Sobel operator. This step helps in assessing the rate of change in intensity across the image.

We then compute the second-order derivatives gxx, gyy, and gxy (cross-derivative). These derivatives are essential to understand the curvature of the image intensity landscape at each point. The strength of saddle points is computed using the formula (S = gxx * gyy - gxy^2). This formula identifies locations where the intensity surface curves upwards in one direction and downwards in another, indicating a saddle point.



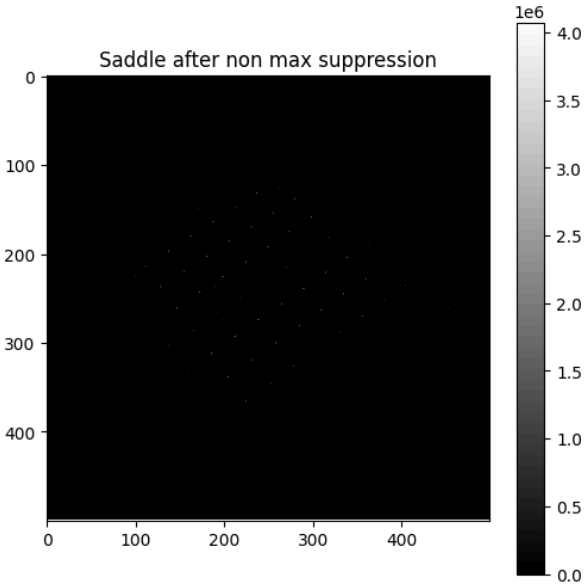
Non-Maximum Suppression

Non-maximum suppression is a crucial step in edge detection, used to refine the edges to ensure they are sharp and thin. In our chessboard recognition system, non-maximum suppression helps in clearly defining the boundaries of chess pieces and the edges of the chessboard.

We start by setting up an output image array `img_sup`, initialized to zeros and of the same shape and type as the input image. This array will store the result of the non-maximum suppression process. For each pixel that has a non-zero value in the input image (indicating potential edge points), we examine a neighborhood around the pixel. This neighborhood is defined by a window of size defined by a parameter.

Within the defined window, we compare the value of the current pixel with the maximum value found in its neighborhood. If the current pixel's value is equal to the maximum value and it is the only pixel in that neighborhood with this maximum value (ensuring uniqueness), it is preserved in the output image `img_sup`. The output of this step is an image with only the strongest edges being retained, effectively thinning out weaker and less significant edges.

Thus by implementing non-maximum suppression, we enhance the clarity and precision of the edge maps produced during the initial stages of our image processing.



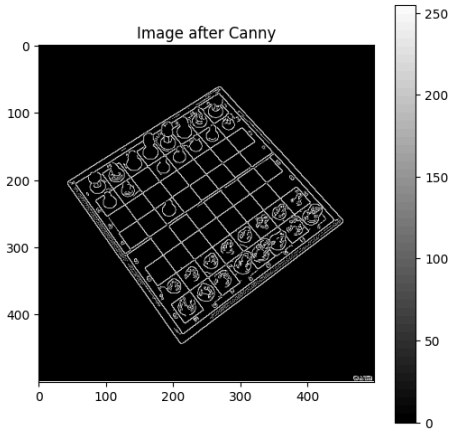
Pruning Saddle Points

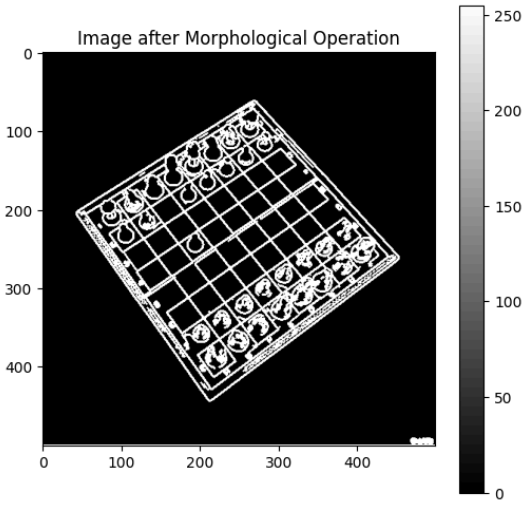
Next, we employ a pruning method specifically designed for saddle points to enhance the clarity and relevance of the detected features. This pruning process helps in focusing on significant saddle points while reducing noise, which is crucial for accurate chess piece recognition and move analysis.

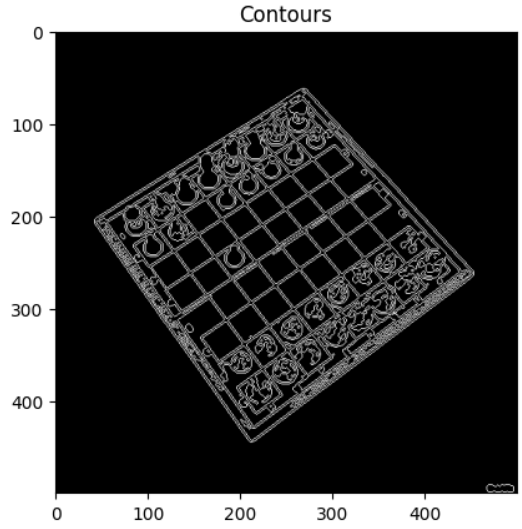
We start by counting the initial number of features (saddle points) whose strength is above zero, providing a baseline of how many features are initially considered. The core of the pruning process involves dynamically adjusting the threshold to gradually eliminate less significant features. The initial threshold is used to suppress any saddle point whose strength is below this value. If the total number of features still exceeds the `max_features` limit, the threshold is doubled. This process is repeated, doubling the threshold and suppressing features below the new threshold, until the number of features is reduced to or below the `max_features` limit. Values below the current threshold are set to zero, effectively pruning weaker saddle points from consideration.

Contour Simplification Using the Douglas-Peucker Algorithm

Next, we use the Canny edge detection algorithm provided by OpenCV library (‘cv2.Canny()’) to find edges and contours in the image and extract contours from the image using morphological operations to highlight them. Using OpenCV’s `cv2.findContours` function, contours are extracted from the morphologically enhanced edges. Simplifying detected contours is essential for efficient and accurate representation of shapes, particularly the squares of the chessboard and the outlines of chess pieces. For this, we use the Douglas-Peucker algorithm to reduce the complexity of these contours while preserving their essential geometric structure.

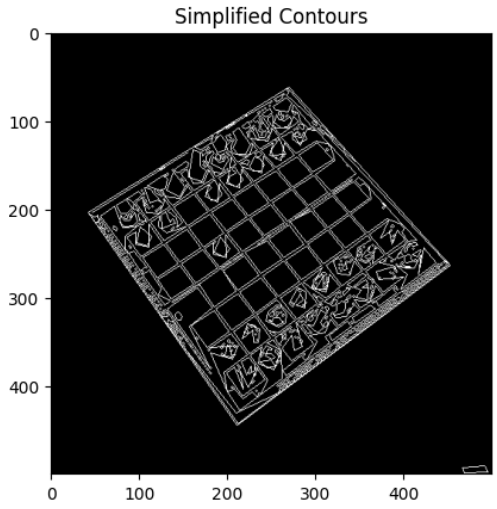






Iterating through the contours, the parameter `epsilon` is calculated as a fraction (in this case, 4%) of the contour's total arc length for each contour. The `cv2.approxPolyDP` function is used to simplify each contour. This function takes the original contour and the `epsilon` value and returns a new contour that approximates the original with fewer points.

By reducing the number of points in each contour, the computational load in subsequent processing stages is decreased, enhancing overall system performance. Simplified contours are generally easier to analyze and classify, as irrelevant details are removed, and the essential shape is retained.



Angle Calculation Using the Cosine Rule

Next is accurately calculating the angle between chess pieces on the chessboard for determining the orientation of pieces and can assist in validating moves by ensuring that pieces move in geometrically consistent patterns, adhering to the rules of chess. For this, we use the cosine rule. The cosine rule is a fundamental principle in triangle geometry that relates the lengths of the sides of a triangle to the cosine of one of its angles. In mathematical terms, for a triangle with sides of lengths a, b and c, and angle theta opposite side c, the cosine rule states:

c**2 = a**2 + b**2 - 2*a*b*cos(theta)

Rearranging the formula to solve for cos(theta), we obtain:

cos(theta) = \frac{a**2 + b**2 - c*2}{2ab}

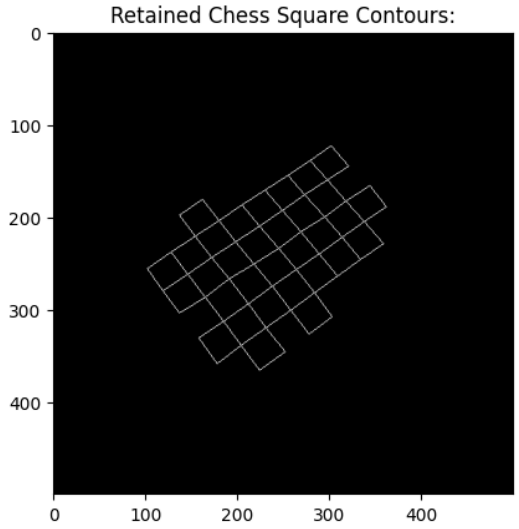
Square Detection in Contours

Next is accurately determining whether a contour corresponds to a square shape, such as those defining the individual squares of a chessboard. We use geometric criteria involving side lengths, diagonal ratios, and angles to assess if a contour is approximately square.

We calculate the Euclidean distances between successive points in the contour to determine the lengths of the sides. This is done using the numpy functions for square and sum to handle the calculations for all sides at once efficiently. The lengths of the diagonals are calculated using the norm of the difference between appropriate contour points, specifically the first and third points for one diagonal, and the second and fourth points for the other.

Next is angle calculation, where for each corner of the contour, the angle is calculated with the appropriate side lengths. This involves calculating angles for all four corners of the contour.

The contour is considered to approximate a square if it meets all the criteria: side ratios within tolerance, acceptable diagonal ratio, and angles indicative of a square. If any of these criteria are not met, the contour is not classified as a square. We compute the centroids of each square, providing useful geometric data for further processing or game analysis.

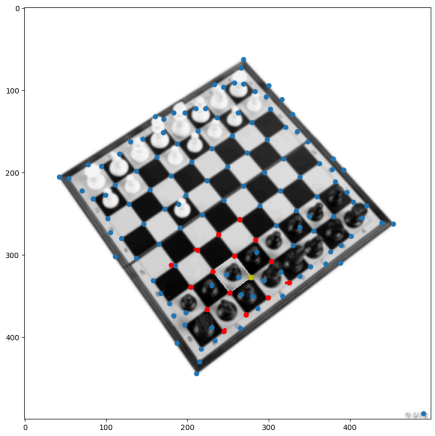


Corner Update Using Saddle Points

The next step is to accurately refine the corners of detected contours for precise boundary definition, especially when analyzing the geometry of chessboard squares or piece placements. We use saddle points to refine these corners.

For each corner within the contour, we extract the coordinates of each corner and define a search window around it. We then identify the strongest saddle point within the

window by locating the maximum value in the saddle point matrix. Upon finding a valid saddle point, we then adjust the corner's position to match the saddle point's location. If no significant saddle point is found, the original corner position is retained to ensure stability and avoid unnecessary adjustments.



Perspective Transformation and Chess Grid Generation

Next, we calculate the perspective transformation needed to map a quadrilateral to a normalized grid, aiding in the recognition of chess squares or board layout under varying viewpoints. This process is crucial for accurate chessboard analysis in images taken from different angles or distances. We create two arrays, representing the row and column indices respectively. A meshgrid is then formed from these indices, which is subsequently flattened and combined to produce a list of grid coordinates. This identity grid acts as the target layout for the perspective transformation, representing a standardized view of a chessboard.

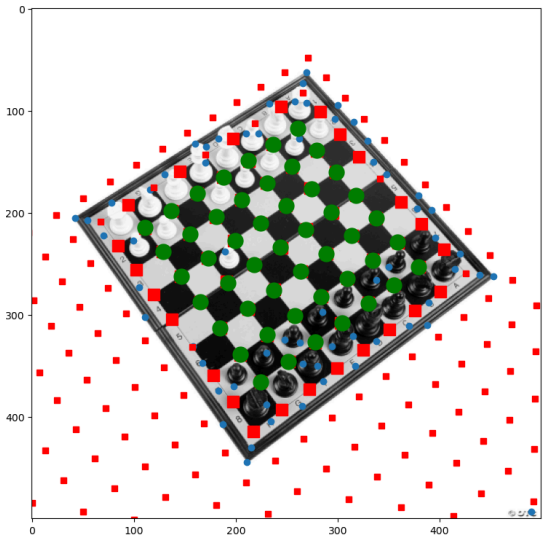
Then, we use the OpenCV function cv2.getPerspectiveTransform to compute the transformation matrix. This function takes the predefined quadrilateral and the input quadrilateral to calculate the matrix M. This matrix transforms the unit square to match the given quadrilateral, effectively mapping the standardized grid points to the actual points on the image. Finally, transformation is applied using matrix multiplication, and then normalized by dividing by the third coordinate, aligning the grid to the image's perspective and scaling it to a usable 2D coordinate system.

Saddle Point Distance Calculation and Grid Adjustment

Achieving precise grid alignment with actual features on the chessboard image is essential for accurate analysis. We calculate the nearest saddle point to a given grid point. This is done by evaluating the Euclidean distance squared between each saddle point and the grid point, selecting the saddle point with the minimal distance. Next, we copy the original grid, initializing a set to track which saddle points are chosen, and using a boolean array to indicate which grid points have been successfully adjusted. The actual adjustment involves an iterative process where each grid point is evaluated against the closest saddle point. If a saddle point has already been chosen, the distance is set to the maximum allowable distance to prevent its reselection. The grid points are then snapped to their closest saddle points if the distance is within the allowed range. Thus, we get an adjusted grid alongside a boolean array that indicates which grid points were successfully aligned to a saddle point within the permitted distance.

Chessboard Grid Initialization

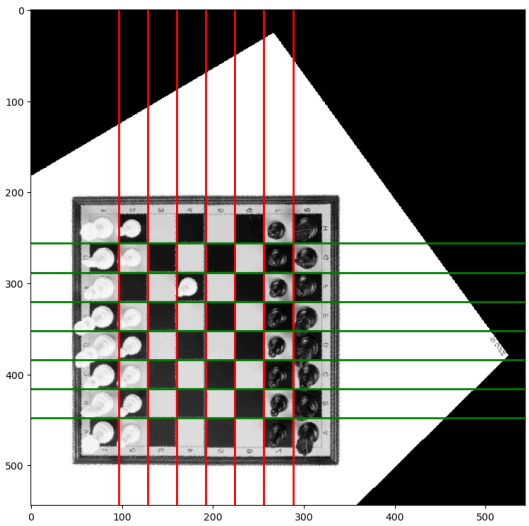
Next is chessboard grid initialization and refinement to ensure that the system can accurately map and adapt to the actual perspective and orientation of the chessboard in the image. We use a specified quadrilateral, which represents a rough estimate of the chessboard's perspective in the image, to initialize the chessboard grid. A perspective transformation is then applied to convert the unit square coordinates into the coordinates provided by quad. This transformation aligns the grid with the perceived orientation and shape of the chessboard as it appears in the image. We then generate a detailed grid based on the transformation matrix M. This grid is expanded to include N additional layers of points around the central four points, which improves the grid's coverage and allows for a more detailed analysis at the boundaries of the chessboard.

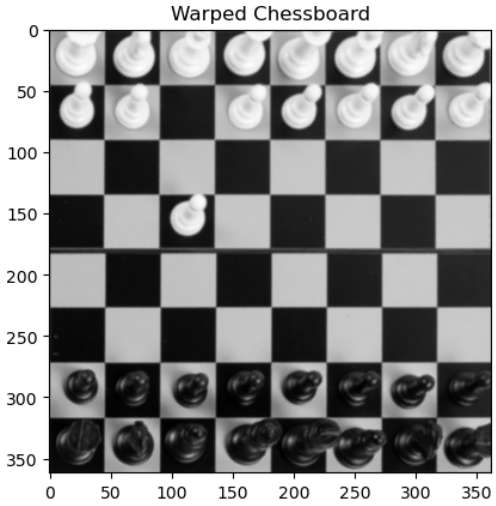


Then, we calculate a new homography that optimally fits the ideally placed grid points to the adjusted points confirmed as good, which align well with detected features such as saddle points. The homography calculation employs the RANSAC algorithm to robustly estimate a homography that minimizes discrepancies between ideally placed grid points and the actual detected good grid points.

Refine Homography and Image Alignment

Then, we calculate a refined homography matrix using a set of corresponding internal points (beyond corners) from two images of a chessboard, using the RANSAC method to handle outliers. This matrix is crucial for aligning the two images precisely. Then transform a set of points based on the refined homography matrix, converting these points from one image's coordinate system to another. This step is essential for comparing features directly between the two images. Then apply the homography matrix to warp one image to the perspective of another, ensuring that both images can be directly compared. This is particularly useful for aligning features such as grid lines and piece locations.



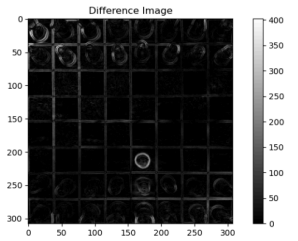


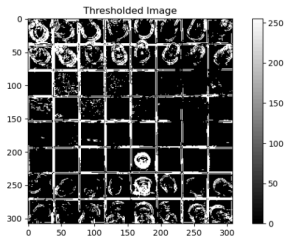
Line Detection and Unwarping

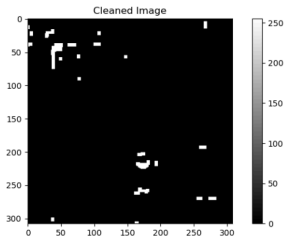
Next, we detect the best grid lines in a warped image of a chessboard and subsequently unwarp these points to their original positions. We start by separating the positive and negative components of the x and y gradients. This separation helps in identifying lines that are significant in both directions (light to dark and dark to light), which are typical for chessboard lines. Then, we compute a score for potential lines by multiplying the sums of positive gradients with the sums of negative gradients across both axes. This scoring method emphasizes lines where there is a strong transition in both directions, which is characteristic of the edges of chessboard squares. From these scores, we select the set of lines that have the highest cumulative scores, indicating the most likely candidates for actual chessboard lines.

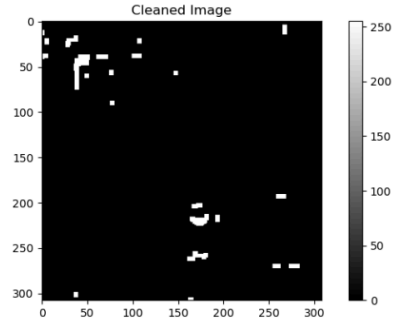
Next, we form a meshgrid from the best line indices detected in the x and y directions. This meshgrid represents the intersection points of the chessboard lines in the warped image. Applying the inverse of the previously computed perspective transformation matrix to these points using `cv2.perspectiveTransform`, we map the points back to their original positions before the image was warped.

Then, we calculate the outer boundary of the chessboard by extrapolating from the best-detected lines, ensuring the entire board is covered. Applies the inverse perspective transformation to map these boundary lines back to their original perspective, providing the actual outline of the chessboard in the scene. Following this, we compute the average spacing between lines to predict where extended lines should be placed, effectively enlarging the grid to include the chessboard's boundary if necessary.









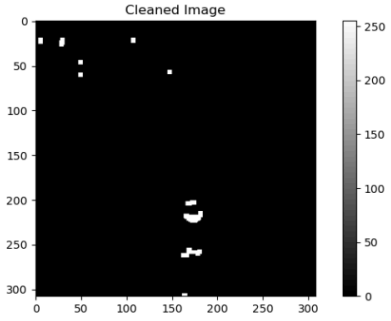


Image Subtraction and Analysis

Next, we remove the grid lines from an image by setting the pixel values along these lines to zero, ensuring that subsequent analyses focus only on the contents of the squares without interference from the grid lines. We iterate through the provided grid points, drawing lines between consecutive points horizontally and vertically using OpenCV's `line` function, setting their pixel values to zero.

We then identify and highlight differences between two images by computing the absolute difference between two images to identify changes, focusing on changes such as chess piece moves. By applying a fixed threshold to the difference image to isolate significant changes, reduce the effect of minor variations like lighting changes or noise.

Using morphological transformations, we apply opening and closing operations to clean the thresholded difference image, removing small noise and enhancing significant features.
Morphological operations are techniques in image processing that focus on the shape or structure within an image. These operations typically involve applying a specific shape or pattern, known as a "structuring element" or "kernel," over the image to process it based on how this kernel fits or intersects with neighboring pixels. The operations you mentioned, opening and closing, are particularly useful for enhancing images by removing noise and filling small holes, respectively.

Morphological Opening (`cv2.MORPH_OPEN`)
The opening operation consists of an erosion followed by a dilation. Here's how it works:

1. Erosion: The initial erosion operation reduces the size of the foreground objects (white regions in a binary image) by removing pixels from the edges. This is useful for removing small noise and separating objects that are close together.

2. Dilation: Following the erosion, a dilation operation is applied, which serves to restore the size of the foreground objects but retains the changes made during the erosion—namely, the removal of small objects and the separation of objects.

```
cleaned = cv2.morphologyEx(thresholded, cv2.MORPH_OPEN, kernel)
```

This operation is particularly useful for eliminating small artifacts and noise from the image, which is essential when you want to focus on significant features like the edges and corners in a chessboard image.

Morphological Closing (`cv2.MORPH_CLOSE`)
The closing operation involves a dilation followed by an erosion. It works as follows:

1. Dilation: This step increases the size of the foreground objects by adding pixels to the boundaries, which helps to close small holes and gaps within the objects.

2. Erosion: The subsequent erosion reduces the size of the objects back to their original size but keeps the holes and gaps filled.

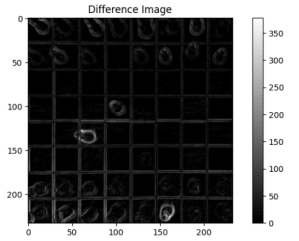
cleaned = cv2.morphologyEx(cleaned, cv2.MORPH_CLOSE, kernel)

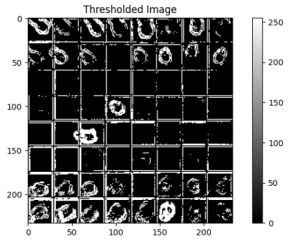
This operation is used to close gaps within detected features and to smooth the outlines of objects. In the context of processing a chessboard image, this could help in ensuring that the chess squares are well-defined and connected, especially useful if the initial thresholding and edge detection leave gaps or breaks in the square boundaries.

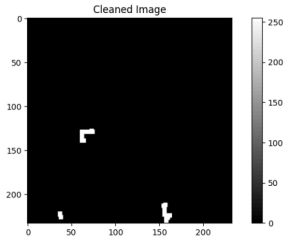
Combined Effect
Using both of these operations sequentially in the image processing pipeline helps achieve a more refined and clean binary image:

- Remove Noise and Small Objects: The opening operation cleans up the image by removing small noise and isolates individual objects.
- Fill Gaps and Smooth Boundaries: The closing operation fills any small holes and smooths the boundaries of objects, making them easier to analyze in subsequent processing stages.

The choice of a 5x5 kernel is a balance between having enough influence to affect meaningful changes in the image without being so large as to overly distort the features. This size is usually sufficient to handle typical noise and small holes in image preprocessing tasks.



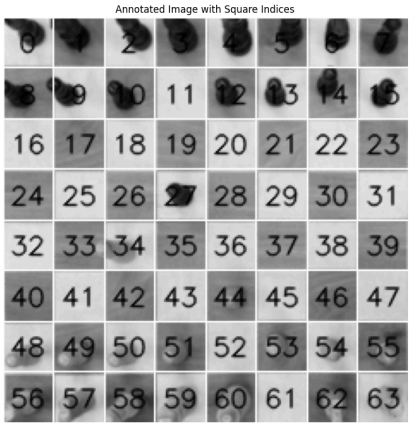


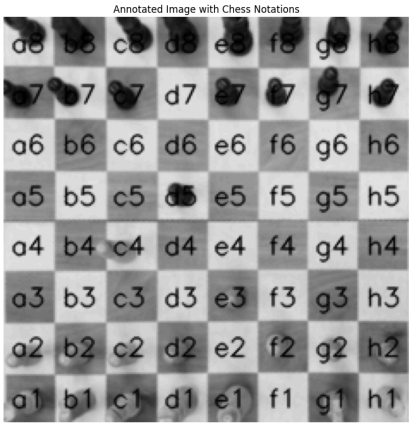


Move detection and Notation

The methodology for translating detected image changes into specific chess moves begins by identifying the most significant alterations within the cleaned image. This process involves isolating and analyzing areas exhibiting the highest change intensities, utilizing image processing techniques such as thresholding and contour detection. Through this step, two distinct regions representing potential chess moves are selected based on their intensity levels, providing a focused basis for subsequent analysis.

Once the top change areas are identified, we proceed to determine the direction of each move. This involves looking at the relative intensities and spatial locations of the selected regions to infer the movement from the starting square to the ending square. By examining the spatial distribution and intensity gradients of the detected changes, the directionality of the move is discerned with precision. This systematic approach not only enables the accurate interpretation of visual alterations but also facilitates the seamless integration of these changes into the chess game, enhancing the overall analysis and strategic decision-making process.





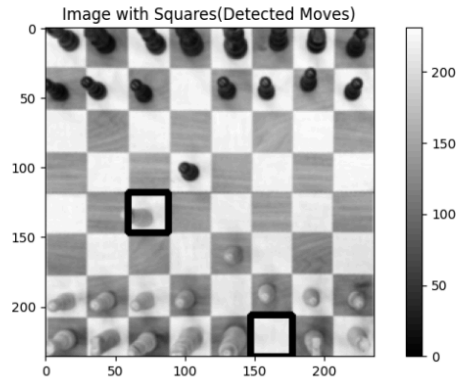
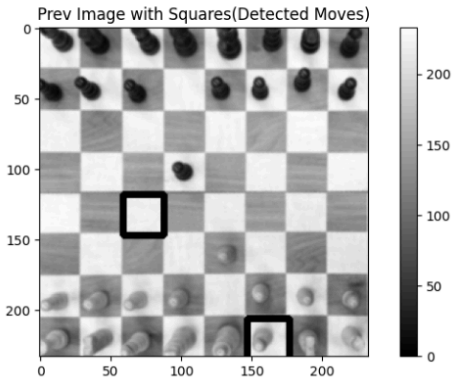
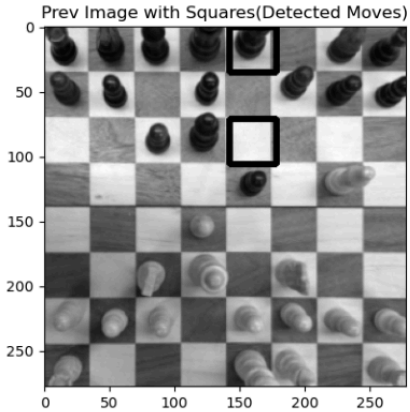


Image with Squares(Detected Moves)
0
50
100
150
200
0
50
100
150
200
0
50
100
150
200


Prev Image with Squares(Detected Moves)
0
50
100
150
200
0
50
100
150
200
0
50
100
150
200


Prev Image with Squares(Detected Moves)
0
50
100
150
200
250
0
50
100
150
200
250

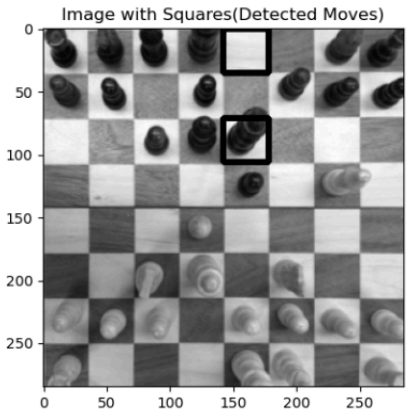


Image with Squares(Detected Moves)
0
50
100
150
200
250
0
50
100
150
200
250

Contributions:

1. Tanmay Rathi - Implemented the logic for the algorithm to detect chessboards and identify moves from images. Conducted experiments and tests using various sets of images to evaluate the algorithm. Identified and rectified errors in the code through debugging processes. Prepared and delivered the presentation for the project. Contributed to capturing images of the chessboard for testing and experimentation purposes.

2. Simardeep Mehta - Designed and implemented the user interface (UI) and backend (Flask) for the web application. Collaborated with Avadhoot to integrate the computer vision algorithm script with the web application. Assisted Tanmay in implementing the logic of the algorithm. Contributed to the documentation and capturing images of the chessboard for testing and experimentation purposes. Jointly delivered the presentation for the project.

3. Avadhoot Kulkarni - Assisted Tanmay in implementing the logic of the algorithm and preparing and delivering the presentation. Wrote the report, providing documentation of the project, including the motivation, methodology, and results. Assisted in testing the code on different sets of images. Assisted Simardeep in integrating the code with the web application. Contributed to capturing images of the chessboard for testing and experimentation purposes.

Conclusion:

In conclusion, our project, "Automated Chess Board Recognition and Interaction using a Top-View Setup," has successfully demonstrated the feasibility and efficiency of applying traditional image processing techniques to the complex problem of real-time chess game monitoring. By opting for a top-view imaging approach, we have streamlined the process of detecting and recognizing chess pieces and their movements, overcoming the challenges posed by multi-angle views and 3D reconstruction typically found in conventional methods.

Our system leverages edge detection, contour extraction, and morphological transformations to accurately interpret the state of the chessboard after each move. The implementation of algorithms like Canny edge detection, Douglas-Peucker contour simplification, and non-maximum suppression has significantly enhanced the precision of our edge and feature detection processes, ensuring that only relevant visual

information is retained. This focus on simplifying image input through strategic preprocessing has allowed for rapid and reliable analysis, which is critical in both competitive and casual chess environments.

Moreover, the incorporation of saddle point detection and advanced image subtraction techniques has further refined our system's capability to detect subtle changes between states, an essential feature for move validation and automated gameplay. By effectively handling real-time data with minimal processing delays, our project offers a scalable and cost-effective alternative to more hardware-intensive setups, such as sensor-equipped boards or deep learning systems that require extensive computational resources.

This project not only fulfills the academic goal of applying learned image processing techniques to a practical scenario but also contributes to the broader field of game analysis technology. Future work could expand on this foundation by integrating machine learning algorithms to predict player strategies or by enhancing the system’s adaptability to various lighting conditions and physical setups.

```
import time
from IPython.display import Image, display
import PIL.Image
import matplotlib.image as mpimg
import scipy.ndimage
import cv2 # For Sobel etc
import glob
import matplotlib.pylab as plt
import numpy as np
np.set_printoptions(suppress=True, linewidth=200)
plt.rcParams['image.cmap'] = 'jet'

import numpy as np
from matplotlib.pyplot import imshow, show
import matplotlib.patches as patches

def compute_gradients(images):

    grad_mags = []
    grad_phases = []

    for img in images:

        gx = cv2.Sobel(img, cv2.CV_64F, 1, 0)
        gy = cv2.Sobel(img, cv2.CV_64F, 0, 1)


        grad_mag = np.sqrt(gx**2 + gy**2).astype(np.float32)
        grad_phase = np.arctan2(gy, gx)


        gradient_mask_threshold = 2 * np.mean(grad_mag.flatten())
        grad_phase_masked = np.where(grad_mag > gradient_mask_threshold, grad_phase, np.nan)


        grad_mags.append(grad_mag)
        grad_phases.append(grad_phase_masked)

    return grad_mags, grad_phases

def getSaddle(gray_img, visualize=False):

    if not isinstance(gray_img, np.ndarray) or len(gray_img.shape) != 2:
        raise ValueError("Input must be a single-channel grayscale image.")

    img = gray_img.astype(np.float64)
    gx = cv2.Sobel(img, cv2.CV_64F, 1, 0)
    gy = cv2.Sobel(img, cv2.CV_64F, 0, 1)
    gxx = cv2.Sobel(gx, cv2.CV_64F, 1, 0)
    gyy = cv2.Sobel(gy, cv2.CV_64F, 0, 1)
    gxy = cv2.Sobel(gx, cv2.CV_64F, 0, 1)

    S = gxx * gyy - gxy**2

    if visualize:
        plt.figure(figsize=(6, 6))
        plt.imshow(S, cmap='jet')
        plt.colorbar()
        plt.title("Saddle Points")
        plt.show()

    return S

def nonmax_sup(img, win=10):

    w, h = img.shape
    img_sup = np.zeros_like(img, dtype=np.float64)
```

```
    for i, j in np.argwhere(img):

        ta = max(0, i - win)
        tb = min(w, i + win + 1)
        tc = max(0, j - win)
        td = min(h, j + win + 1)
        cell = img[ta:tb, tc:td]
        val = img[i, j]


        if (cell.max() == val and np.sum(cell.max() == cell) == 1):
            img_sup[i, j] = val

    return img_sup

def pruneSaddle(s, max_features=10000, initial_thresh=128):

    thresh = initial_thresh
    score = (s > 0).sum()

    while score > max_features:
        thresh *= 2
        s[s < thresh] = 0
        score = (s > 0).sum()



def process_saddles(images):

    saddles = []
    for img in images:
        saddle = getSaddle(img)
        saddle = np.maximum(-saddle, 0)

        pruneSaddle(saddle)

        saddles.append(saddle)


        plt.figure(figsize=(6, 6))
        plt.imshow(saddle, cmap='gray')
        plt.title("Pruned Saddle Points")
        plt.colorbar()
        plt.show()

    return saddles

def simplifyContours(contours):

    for i in range(len(contours)):
        epsilon = 0.04 * cv2.arcLength(contours[i], True)
        contours[i] = cv2.approxPolyDP(contours[i], epsilon, True)

def getAngle(a, b, c):


    k = (a*a + b*b - c*c) / (2 * a * b)
    k = max(min(k, 1), -1)
    return np.arccos(k) * (180.0 / np.pi)

def is_square(cnt, eps=3.0, xratio_thresh=0.5):


    dd = np.sqrt(np.sum(np.square(np.diff(cnt[:, 0, :], axis=0, append=cnt[:1, 0, :])),
axis=1))
```

```
    xa = np.linalg.norm(cnt[0, 0, :] - cnt[2, 0, :])
    xb = np.linalg.norm(cnt[1, 0, :] - cnt[3, 0, :])
    xratio = min(xa, xb) / max(xa, xb)


    angles = np.array([getAngle(dd[i], dd[(i+1) % 4], xb if i % 2 == 0 else xa) for i in
range(4)])
    good_angles = np.all((angles > 40) & (angles < 140))


    side_ratios = np.array([max(dd[i] / dd[(i+1) % 4], dd[(i+1) % 4] / dd[i]) for i in
range(4)])
    good_side_ratios = np.all(side_ratios < eps)

    return good_side_ratios and good_angles and xratio > xratio_thresh

def getContourVals(cnt, img):

    cimg = np.zeros_like(img)
    cv2.drawContours(cimg, [cnt], 0, color=255, thickness=-1)
    return img[cimg == 255]

def pruneContours(contours, hierarchy, saddle):

    new_contours, new_hierarchies = [], []
    for i, cnt in enumerate(contours):
        h = hierarchy[i]
        if h[2] != -1 or len(cnt) != 4 or cv2.contourArea(cnt) < 64 or not is_square(cnt):
            continue
        cnt = updateCorners(cnt, saddle)
        if len(cnt) != 4:
            continue
        new_contours.append(cnt)
        new_hierarchies.append(h)

    if not new_contours:
        return np.array([]), np.array([])

    areas = [cv2.contourArea(c) for c in new_contours]
    median_area = np.median(areas)
    filtered_contours = [c for c, a in zip(new_contours, areas) if median_area * 0.25 <= a <=
median_area * 2.0]
    return np.array(filtered_contours), np.array(new_hierarchies)

def getContours(img, edges, iters=10):


    if not isinstance(edges, np.ndarray):
        raise ValueError("Edges input must be a numpy array.")


    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))


    edges_gradient = cv2.morphologyEx(edges, cv2.MORPH_GRADIENT, kernel)


    contours, hierarchy = cv2.findContours(edges_gradient, cv2.RETR_CCOMP,
cv2.CHAIN_APPROX_SIMPLE)


    simplifyContours(contours)


    if hierarchy is None or len(hierarchy) == 0:
        raise ValueError("No hierarchy data found, check the input edges for adequate feature
details.")
```

```
    return np.array(contours), hierarchy[0]

def getContours(img, edges):

    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
    edges_gradient = cv2.morphologyEx(edges, cv2.MORPH_GRADIENT, kernel)
    contours, hierarchy = cv2.findContours(edges_gradient, cv2.RETR_CCOMP,
cv2.CHAIN_APPROX_SIMPLE)


    contours = list(contours)
    simplifyContours(contours)

    return contours, hierarchy[0]

def updateCorners(contour, saddle):
    ws = 4
    new_contour = contour.copy()
    for i in range(len(contour)):
        cc, rr = contour[i, 0, :]

        rl, cl = max(0, rr - ws), max(0, cc - ws)
        rh, ch = min(saddle.shape[0], rr + ws + 1), min(saddle.shape[1], cc + ws + 1)
        window = saddle[rl:rh, cl:ch]

        br, bc = np.unravel_index(window.argmax(), window.shape)
        if window[br, bc] > 0:
            new_contour[i, 0, :] = (cc + bc - min(ws, cl), rr + br - min(ws, rl))
        else:

            return contour
    return new_contour

def getIdentityGrid(N):
    a = np.arange(N)
    b = a.copy()
    aa,bb = np.meshgrid(a,b)
    return np.vstack([aa.flatten(), bb.flatten()]).T

def getChessGrid(quad):
    quadA = np.array([[0,1],[1,1],[1,0],[0,0]],dtype=np.float32)
    M = cv2.getPerspectiveTransform(quadA, quad.astype(np.float32))
    quadB = getIdentityGrid(4)-1
    quadB_pad = np.pad(quadB, ((0,0),(0,1)), 'constant', constant_values=1)
    C_thing = (np.matrix(M)*quadB_pad.T).T

    C_thing[:,:2] /= C_thing[:,2]
    return C_thing

def getMinSaddleDist(saddle_pts, pt):
    best_dist = None
    best_pt = pt
    for saddle_pt in saddle_pts:
        saddle_pt = saddle_pt[::-1]
        dist = np.sum((saddle_pt - pt)**2)
        if best_dist is None or dist < best_dist:
            best_dist = dist
            best_pt = saddle_pt
    return best_pt, np.sqrt(best_dist)


def findGoodPoints(grid, spts, max_px_dist=5):

    new_grid = grid.copy()
    chosen_spts = set()
    N = len(new_grid)
```

```
    grid_good = np.zeros(N,dtype=np.bool_)
    hash_pt = lambda pt: "%d_%d" % (pt[0], pt[1])

    for pt_i in range(N):
        pt2, d = getMinSaddleDist(spts, grid[pt_i,:2].A.flatten())
        if hash_pt(pt2) in chosen_spts:
            d = max_px_dist
        else:
            chosen_spts.add(hash_pt(pt2))
        if (d < max_px_dist):
            new_grid[pt_i,:2] = pt2
            grid_good[pt_i] = True
    return new_grid, grid_good

def getInitChessGrid(quad):
    quadA = np.array([[0,1],[1,1],[1,0],[0,0]],dtype=np.float32)
    M = cv2.getPerspectiveTransform(quadA, quad.astype(np.float32))
    return makeChessGrid(M,1)

def makeChessGrid(M, N=1):
    ideal_grid = getIdentityGrid(2+2*N)-N
    ideal_grid_pad = np.pad(ideal_grid, ((0,0),(0,1)), 'constant', constant_values=1)

    grid = (np.matrix(M)*ideal_grid_pad.T).T
    grid[:,:2] /= grid[:,2]
    grid = grid[:,:2]
    return grid, ideal_grid, M

def generateNewBestFit(grid_ideal, grid, grid_good):
    a = np.float32(grid_ideal[grid_good])
    b = np.float32(grid[grid_good])
    M = cv2.findHomography(a, b, cv2.RANSAC)
    return M

def getGrads(img):
    img = cv2.blur(img,(5,5))
    gx = cv2.Sobel(img,cv2.CV_64F,1,0)
    gy = cv2.Sobel(img,cv2.CV_64F,0,1)

    grad_mag = gx*gx+gy*gy
    grad_phase = np.arctan2(gy, gx)
    grad_phase_masked = grad_phase.copy()
    gradient_mask_threshold = 2*np.mean(grad_mag.flatten())
    grad_phase_masked[grad_mag < gradient_mask_threshold] = np.nan
    return grad_mag, grad_phase_masked, grad_phase, gx, gy



def getBestLines(img_warped):
    grad_mag, grad_phase_masked, grad_phase, gx, gy = getGrads(img_warped)

    # X
    gx_pos = gx.copy()
    gx_pos[gx_pos < 0] = 0
    gx_neg = -gx.copy()
    gx_neg[gx_neg < 0] = 0
    score_x = np.sum(gx_pos, axis=0) * np.sum(gx_neg, axis=0)
    # Y
    gy_pos = gy.copy()
    gy_pos[gy_pos < 0] = 0
    gy_neg = -gy.copy()
    gy_neg[gy_neg < 0] = 0
    score_y = np.sum(gy_pos, axis=1) * np.sum(gy_neg, axis=1)


    a = np.array([(offset + np.arange(7) + 1)*32 for offset in np.arange(1,11-2)])
    scores_x = np.array([np.sum(score_x[pts]) for pts in a])
```

```
    scores_y = np.array([np.sum(score_y[pts]) for pts in a])


    best_lines_x = a[scores_x.argmax()]
    best_lines_y = a[scores_y.argmax()]
    return (best_lines_x, best_lines_y)

def getUnwarpedPoints(best_lines_x, best_lines_y, M):
    x,y = np.meshgrid(best_lines_x, best_lines_y)
    xy = np.vstack([x.flatten(), y.flatten()]).T.astype(np.float32)
    xy = np.expand_dims(xy,0)

    xy_unwarp = cv2.perspectiveTransform(xy, M)
    return xy_unwarp[0,:,:]

def loadImage(image_path, isRotated):
    img_orig = PIL.Image.open(image_path)
    img_width, img_height = img_orig.size
    print(f'In load class, value {isRotated}')
    if not isRotated:
        img_orig = img_orig.rotate(90)


    aspect_ratio = min(500.0/img_width, 500.0/img_height)
    new_width, new_height = ((np.array(img_orig.size) * aspect_ratio)).astype(int)
    img = img_orig.resize((new_width,new_height), resample=PIL.Image.BILINEAR)
    img = img.convert('L')
    img = np.array(img)

    return img

def findChessboard(img, min_pts_needed=15, max_pts_needed=25):
    blur_img = cv2.blur(img, (3,3))
    saddle = getSaddle(blur_img)
    saddle = -saddle
    saddle[saddle<0] = 0
    pruneSaddle(saddle)
    s2 = nonmax_sup(saddle)
    s2[s2<100000]=0
    spts = np.argwhere(s2)

    edges = cv2.Canny(img, 20, 250)
    contours_all, hierarchy = getContours(img, edges)
    contours, hierarchy = pruneContours(contours_all, hierarchy, saddle)

    curr_num_good = 0
    curr_grid_next = None
    curr_grid_good = None
    curr_M = None

    for cnt_i in range(len(contours)):

        cnt = contours[cnt_i].squeeze()
        grid_curr, ideal_grid, M = getInitChessGrid(cnt)

        for grid_i in range(7):
            grid_curr, ideal_grid, _ = makeChessGrid(M, N=(grid_i+1))
            grid_next, grid_good = findGoodPoints(grid_curr, spts)
            num_good = np.sum(grid_good)

            if num_good < 4:
                M = None

                break
            M, _ = generateNewBestFit(ideal_grid, grid_next, grid_good)

            if M is None or np.abs(M[0,0] / M[1,1]) > 15 or np.abs(M[1,1] / M[0,0]) > 15:
```

```
                M = None
                #print ("Failed to converge on this one")
                break
        if M is None:
            continue
        elif num_good > curr_num_good:
            curr_num_good = num_good
            curr_grid_next = grid_next
            curr_grid_good = grid_good
            curr_M = M


        if num_good > max_pts_needed:
            break


    if curr_num_good > min_pts_needed:
        final_ideal_grid = getIdentityGrid(2+2*7)-7
        return curr_M, final_ideal_grid, curr_grid_next, curr_grid_good, spts
    else:
        return None, None, None, None, None
#     return M, ideal_grid, grid_next, grid_good, spts

def getBoardOutline(best_lines_x, best_lines_y, M):
    d = best_lines_x[1] - best_lines_x[0]
    ax = [best_lines_x[0]-d, best_lines_x[-1]+d]
    ay = [best_lines_y[0]-d, best_lines_y[-1]+d]
    x,y = np.meshgrid(ax, ay)
    xy = np.vstack([x.flatten(), y.flatten()]).T.astype(np.float32)
    xy = xy[[0,1,3,2,0],:]
    xy = np.expand_dims(xy,0)

    xy_unwarp = cv2.perspectiveTransform(xy, M)
    return xy_unwarp[0,:,:]

def extrapolate_lines(lines):
    spacings = np.diff(lines)
    avg_spacing = np.mean(spacings)
    extended_lines = np.zeros(len(lines) + 2)
    extended_lines[1:-1] = lines
    extended_lines[0] = lines[0] - avg_spacing
    extended_lines[-1] = lines[-1] + avg_spacing
    return extended_lines

def compute_grid_points(size, grid_count=9):

    step = size / (grid_count - 1)
    grid_points = np.array([(x * step, y * step) for y in range(grid_count) for x in
range(grid_count)])
    return grid_points.reshape((grid_count, grid_count, 2))

def get_squares_from_points(points):
    squares = []
    for i in range(points.shape[0] - 1):
        for j in range(points.shape[1] - 1):
            top_left = points[i, j]
            top_right = points[i, j+1]
            bottom_right = points[i+1, j+1]
            bottom_left = points[i+1, j]
            square = [top_left, top_right, bottom_right, bottom_left]
            squares.append(square)
    return squares

def extract_templates(image, squares):

    templates = []
```

```
    for square in squares:

        pts = np.array(square, np.int32)
        rect = cv2.boundingRect(pts)
        x, y, w, h = rect
        template = image[y:y+h, x:x+w]
        templates.append(template)
    return templates

def calculate_centroid(points):

        points = np.array(points)
        x = np.mean(points[:, 0])
        y = np.mean(points[:, 1])
        return (x, y)

def get_square_centroids(squares):

    square_centroids = [calculate_centroid(square) for square in squares]
    return square_centroids

def process_chessboard_image(image_path, isRotated):
    image = loadImage(image_path, isRotated)
    matrix, ideal_grid, grid_next, grid_good, spts = findChessboard(image)

    if matrix is not None:
        matrix, _ = generateNewBestFit((ideal_grid+8)*32, grid_next, grid_good)
        img_warp = cv2.warpPerspective(image, matrix, (17*32, 17*32),
flags=cv2.WARP_INVERSE_MAP)

        lines_x, lines_y = getBestLines(img_warp)
        xy_unwarp = getUnwarpedPoints(lines_x, lines_y, matrix)

    else:
        print("Matrix is None. Fail")
        return None

    lines_x_extended = extrapolate_lines(lines_x)
    lines_y_extended = extrapolate_lines(lines_y)
    xy_unwarp_extended = getUnwarpedPoints(lines_x_extended, lines_y_extended, matrix)
    points_grid = xy_unwarp_extended.reshape((9, 9, 2))

    squares = []
    for i in range(8):
        for j in range(8):
            top_left = points_grid[i, j]
            top_right = points_grid[i, j + 1]
            bottom_right = points_grid[i + 1, j + 1]
            bottom_left = points_grid[i + 1, j]
            square = [top_left, top_right, bottom_right, bottom_left]
            squares.append(square)


    top_left, bottom_left, bottom_right, top_right = points_grid[0, 0], points_grid[0, -1],
points_grid[-1, -1], points_grid[-1, 0]
    src_points = np.array([top_left, top_right, bottom_right, bottom_left], dtype=np.float32)

    size = max(
        np.linalg.norm(top_right - top_left),
        np.linalg.norm(bottom_right - top_right),
        np.linalg.norm(bottom_left - top_left),
        np.linalg.norm(bottom_right - bottom_left)
    )
    dest_points = np.array([
        [0, 0],
        [size, 0],
        [size, size],
```

```
        [0, size]
    ], dtype=np.float32)

    matrix = cv2.getPerspectiveTransform(src_points, dest_points)
    warped_image = cv2.warpPerspective(image, matrix, (int(size), int(size)))
    grad_mag_image, _ = compute_gradients([warped_image])
    grad_mag_image = grad_mag_image[0]

    new_grid_points = compute_grid_points(int(size))
    new_squares = get_squares_from_points(new_grid_points)
    square_centroids = get_square_centroids(new_squares)

    return {
        'warped_image': warped_image,
        'new_squares': new_squares,
        'new_grid_points': new_grid_points,
        'transformation_matrix': matrix,
        'square_centroids': square_centroids,
        'lines_x': lines_x,
        'lines_y': lines_y,
        'xy_unwarp': xy_unwarp,
        'lines_x_extended': lines_x_extended,
        'lines_y_extended': lines_y_extended,
        'xy_unwarp_extended': xy_unwarp_extended,
        'squares': squares,
        'src_points': src_points,
        'size': size,
        'dest_points': dest_points,
        'grad_mag_image': grad_mag_image
    }

def zero_out_lines(image, grid_points):

    for i in range(len(grid_points)):
        for j in range(len(grid_points[0])):
            if j < len(grid_points[0]) - 1:
                cv2.line(image, tuple(grid_points[i][j].astype(int)), tuple(grid_points[i]
[j+1].astype(int)), 0, 1)
            if i < len(grid_points) - 1:
                cv2.line(image, tuple(grid_points[i][j].astype(int)), tuple(grid_points[i+1]
[j].astype(int)), 0, 1)

def apply_multiscale_morphological_operations(image, kernel_sizes):
    combined_result = None
    for size in kernel_sizes:
        kernel = np.ones((size, size), np.uint8)

        opened = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
        closed = cv2.morphologyEx(opened, cv2.MORPH_CLOSE, kernel)

        if combined_result is None:
            combined_result = closed
        else:
            combined_result = cv2.bitwise_or(combined_result, closed)

    return combined_result

def advanced_morphological_operations(image):

    kernel = np.ones((3,3), np.uint8)

    dilated = cv2.dilate(image, kernel, iterations=1)


    combined = cv2.bitwise_or(image, dilated)
```

```
    closed = cv2.morphologyEx(combined, cv2.MORPH_CLOSE, kernel)
    return closed

def subtract_images(artifacts, image2):
    image1 = artifacts['grad_mag_image']
    grid_points = artifacts['new_grid_points']

    img1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY) if len(image1.shape) == 3 else image1
    img2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY) if len(image2.shape) == 3 else image2


    difference = cv2.absdiff(img1, img2)


    _, thresholded = cv2.threshold(difference, 30, 255, cv2.THRESH_BINARY)



    if grid_points is not None:
        zero_out_lines(thresholded, grid_points)


    thresholded = thresholded.astype(np.uint8)


    kernel = np.ones((5,5), np.uint8)
    cleaned = cv2.morphologyEx(thresholded, cv2.MORPH_OPEN, kernel)
    cleaned = cv2.morphologyEx(cleaned, cv2.MORPH_CLOSE, kernel)

    num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(cleaned,
connectivity=8)


    return difference, thresholded, cleaned

def refine_homography_with_internal_points(frame1_points, frame2_points):

    H, status = cv2.findHomography(frame2_points, frame1_points, cv2.RANSAC)
    return H

def apply_homography(H, points):

    points_homogeneous = np.hstack([points, np.ones((points.shape[0], 1))])
    transformed_points = np.dot(H, points_homogeneous.T).T
    transformed_points /= transformed_points[:, 2][:, np.newaxis]
    return transformed_points[:, :2]

def warp_image(image, H, size):
    warped_image = cv2.warpPerspective(image, H, (size[1], size[0]))
    return warped_image

def align_and_compare_images(artifacts_1, artifacts_2):

    grid_points_frame1 = np.array(artifacts_1['new_grid_points'].reshape(81, 2))
    grid_points_frame2 = np.array(artifacts_2['new_grid_points'].reshape(81, 2))


    refined_homography = refine_homography_with_internal_points(grid_points_frame1,
grid_points_frame2)


    aligned_points_frame2 = apply_homography(refined_homography, grid_points_frame2)


    displacements = grid_points_frame1 - aligned_points_frame2
```

```
    dimensions = (artifacts_1['grad_mag_image'].shape[1],
artifacts_1['grad_mag_image'].shape[0])
    aligned_grad_mag_image_frame2 = warp_image(artifacts_2['grad_mag_image'],
refined_homography, dimensions)

    return displacements, aligned_grad_mag_image_frame2

def compute_square_intensity(square, image):
        mask = np.zeros(image.shape, dtype=np.uint8)
        points = np.array([square], dtype=np.int32)
        cv2.fillPoly(mask, points, 255)
        return np.sum(image[mask == 255])

def get_top_squares(artifacts, thresholded_image):

    square_intensities = [compute_square_intensity(square, thresholded_image) for square in
artifacts['new_squares']]


    top_two_indices = np.argsort(square_intensities)[-2:]

    top_two_squares = [artifacts['new_squares'][i] for i in top_two_indices]
    top_two_intensities = [square_intensities[i] for i in top_two_indices]

    return top_two_indices, top_two_squares, top_two_intensities

def index_to_chess_notation(index):
    file = chr((index % 8) + ord('a'))
    rank = 8 - (index // 8)
    return f"{file}{rank}"

def determine_move_direction(artifacts_1, index1, index2):
    grad_image = artifacts_1['grad_mag_image']
    square1 = artifacts_1['new_squares'][index1]
    square2 = artifacts_1['new_squares'][index2]

    avg_intensity1 = np.mean(square1)
    avg_intensity2 = np.mean(square2)

    if avg_intensity1 > avg_intensity2:
        move_start_idx, move_end_idx = index1, index2
    else:
        move_start_idx, move_end_idx = index2, index1

    return move_start_idx, move_end_idx

def nextMove(image_path_1, image_path_2, isRotated=False):
    start_time = time.time()

    artifacts_1 = process_chessboard_image(image_path_1, isRotated)
    artifacts_2 = process_chessboard_image(image_path_2, isRotated)

    displacements, aligned_grad_mag_image_frame2 = align_and_compare_images(artifacts_1,
artifacts_2)
    difference_image, thresholded_image, cleaned_image = subtract_images(artifacts_1,
aligned_grad_mag_image_frame2)

    top_two_indices, top_two_squares, top_two_intensities = get_top_squares(artifacts_2,
cleaned_image)
    move_start_idx, move_end_idx = determine_move_direction(artifacts_1, top_two_indices[0],
top_two_indices[1])


    start_notation = index_to_chess_notation(move_start_idx)
    end_notation = index_to_chess_notation(move_end_idx)

    end_time = time.time()
```

```
    print(f"Elapsed time: {end_time - start_time} seconds")
    return [start_notation, end_notation]
```

```python
from flask import Flask, render_template, session, request, redirect, url_for
import chess
import chess.svg
import os
import time
import helper

app = Flask(__name__)
app.config['SECRET_KEY'] = 'daf18fc78a01e813'
#IMAGE_DIR= 'C:/Users/mehta/Downloads/ArchiveFINAL'
IMAGE_DIR = 'C:/Users/mehta/Downloads/chessmoves'
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}

def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

def process_move(board, src_dest):
    try:
        move_obj = chess.Move.from_uci(src_dest[0] + src_dest[1])
        if move_obj in board.legal_moves:
            board.push(move_obj)
            return True
    except ValueError:
        pass

    try:
        move_obj_reversed = chess.Move.from_uci(src_dest[1] + src_dest[0])
        if move_obj_reversed in board.legal_moves:
            board.push(move_obj_reversed)
            return True
    except ValueError:
        pass

    return False

@app.route('/', methods=['GET', 'POST'])
def index():
    if 'board' not in session or 'initialize' in request.form:
        session['board'] = chess.Board().fen()
        session['index'] = 0
        session['rotate'] = request.form.get('rotate') == 'on'
        files = sorted([f for f in os.listdir(IMAGE_DIR) if allowed_file(f)])
        session['files'] = files if files else []

    board = chess.Board(session['board'])
    files = session.get('files', [])
    index = session.get('index', 0)
    rotate_value = session.get('rotate', False)

    if request.method == 'POST':
        if 'move' in request.form and len(files) > index + 1:
            start_time = time.time()
            image_path_1 = os.path.join(IMAGE_DIR, files[index])
            image_path_2 = os.path.join(IMAGE_DIR, files[index + 1])
            move = helper.nextMove(image_path_1, image_path_2, rotate_value)
            if process_move(board, move):
                session['board'] = board.fen()
                session['index'] = (index + 1) % len(files)
                elapsed_time = time.time() - start_time
                session['elapsed_time'] = float('%.2f' % elapsed_time)
            else:
                print("Invalid move attempted")

    return render_template('index.html', board_svg=chess.svg.board(board), fen=board.fen(),
                           elapsed_time=session.get('elapsed_time'),
rotate=session.get('rotate', False))
```

```python
@app.route('/reset-stock', methods=['POST'])
def reset_stock():
    session['board'] = chess.Board().fen()
    session['index'] = 0
    session.pop('elapsed_time', None)
    session['rotate'] = False  # Reset rotation to default when board is reset
    return redirect(url_for('index'))

if __name__ == '__main__':
    app.run(debug=True)
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chess Board Viewer</title>

    <link rel="stylesheet" href="{{ url_for('static', filename='styles.css')
}}">
    <style>
        .loader {
            border: 16px solid #f3f3f3;
            border-top: 16px solid #5d534a;
            border-radius: 50%;
            width: 100px;
            height: 100px;
            animation: spin 2s linear infinite;
            position: fixed; /* Fixed position */
            left: 50%;
            top: 10px;
            transform: translateX(-50%);
        }

        @keyframes spin {
            0% { transform: rotate(0deg); }
            100% { transform: rotate(360deg); }
        }
    </style>
    <script>
        function showSpinner() {
            document.getElementById('spinner').style.display = 'block';
        }

        window.onload = function() {
            document.getElementById('spinner').style.display = 'none';
        }
    </script>
</head>
<body>
    <div class="header">
        <h1>IdentifyChess - Chess Board Analyzer</h1>
    </div>

    <div class="container">
        <div class="board-container">
            {{ board_svg|safe }}
            {% if image_path %}
                <img src="{{ image_path }}" alt="Initial Chess Position"
style="max-width: 100%; height: auto;">
            {% endif %}
        </div>
        <div class="info-container">
            <h1>Chess Game Controls</h1>

            <form action="/" method="post">
                <!-- <form action="/" method="post"> -->
```

```
                <input type="checkbox" id="rotate" name="rotate" {{ 'checked' if
rotate_value else '' }}>
                <label for="rotate">Rotate</label>
                <input type="submit" name="initialize" value="Initialize">
            </form>
            <form action="/" method="post" onsubmit="showSpinner()">

                <input type="text" id="move" name="move">
                <input type="submit" value="Next Move">
            </form>
            <form action="/reset-stock" method="post">
                <input type="submit" value="Reset Board to Initial Position">
            </form>
            <div id="move-time">
                {% if elapsed_time %}
                    Time Elapsed for Move: {{ "%.2f"|format(elapsed_time) }}
seconds
                {% else %}
                    Time Elapsed for Move: No moves yet
                {% endif %}
            </div>
        </div>

    </div>
    <div id="spinner">
        <div class="loader"></div>
    </div>
</body>
</html>
```

```
/* Global Styles */
* {
    box-sizing: border-box;
    margin: 0;
    padding: 0;
}
html, body {
    height: 100%;
    font-family: 'Arial', sans-serif;
    background-color: #f4f4f3; /* Light cream background for the whole page */
}

/* Container */
.container {
    display: flex;
    flex-wrap: wrap; /* Allow wrapping for smaller screens */
    height: auto; /* Adjust height automatically */
    padding: 20px;
    justify-content: center; /* Center horizontally */
    gap: 20px; /* Space between containers */
}

/* Board Container */
.board-container {
    flex-basis: 600px; /* Adjust width for the chessboard container */
    display: flex;
    justify-content: center;
    align-items: center;
    padding: 20px;
    background-color: #efe0d1; /* Light brown background */
    border: 5px solid #b58863; /* Darker brown border */
    border-radius: 8px;
}

/* Info Container */
.info-container {
    flex-basis: 600px; /* Adjust width for the info container */
    background-color: #ebf5ee; /* Light green background */
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 4px 8px rgba(0,0,0,0.1);
}

/* Typography */
h1 {
    font-family: 'Trebuchet MS', Helvetica, sans-serif; /* Change to a more
stylish font */
    font-size: 28px; /* Increase font size */
    font-weight: bold; /* Make it bold */
    color: #5d534a; /* Dark brown color for text */
    margin-bottom: 20px;
    text-align: center;
     /* Center the heading */
}

/* Form Styles */
```

```
form {
    margin-bottom: 20px;
    display: flex;
    flex-direction: column;
}

label {
    display: block;
    margin-bottom: 10px;
    color: #4a766e; /* Dark green color for labels */
}

input[type="text"], input[type="submit"], input[type="checkbox"] {
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 4px;
    margin-bottom: 15px;
}

input[type="text"] {
    background-color: #fff;
}

input[type="submit"] {
    padding: 15px; /* Larger padding for bigger buttons */
    font-size: 16px; /* Larger font size for text inside the buttons */
    margin-bottom: 25px;
    background-color: #6b8f71; /* Dark green button */
    color: white;
    cursor: pointer;
    border: none; /* Remove default border */
}

input[type="submit"]:hover {

    background-color: #597a5f; /* Slightly darker green for hover */
}

/* Loader Styles */
.loader {
    border-top: 16px solid #6b8f71; /* Dark green to match the button */
}

form {
    margin-bottom: 20px;
    display: flex;
    flex-direction: column;
    align-items: center; /* Align form elements to the center */
}

/* Rotate Checkbox and Label Style */
.rotate-checkbox-container {
    display: flex;
    justify-content: center; /* Center the content horizontally */
    align-items: center; /* Center the content vertically */
    margin-bottom: 15px; /* Spacing below the checkbox container */
```

```
}

.rotate-checkbox-container label {
    margin-left: 10px; /* Space between checkbox and label */
}
/* Header Styles */
.header {
    padding: 20px;
    text-align: center; /* Center the content */
    background-color: #f4f4f3; /* Match the page background */
    margin-bottom: 20px; /* Add some space before the containers */
}
#move-time {
    text-align: center; /* Center the text */
    font-size: 18px; /* Optionally increase the font size */
    padding-top: 10px; /* Add some padding to the top */
    margin-top: 20px; /* Add some margin to the top */
    border-top: 1px solid #ccc; /* Add a separator line */
}



/* Responsive Design Adjustments */
@media (max-width: 768px) {
    .container {
        flex-direction: column;
    }

    .board-container,
    .info-container {
        flex-basis: auto; /* Take full width on small screens */
        width: 100%;
        margin-bottom: 20px; /* Add space between containers */
    }
}
```