Computer Vision - Project 1 / Mathable score calculator

Sîrbu Oana-Adriana
407 AI

1 Defining Paths and Variables

1.1 Base Directory

The `base_dir` variable points to the main folder, which contains a folder with images of the games (train or test images along with the annotated data stored in .txt files) and result folders (processed images, final .txt files generated). Users should adjust this path according to their directory structure (what is their working directory).

1.2 Image Paths

- `base_image_path`: Path to the board image provided for the project.
- `tokens_image_path`: Path to the tokens image provided for the project.
- `template_path`: Path to the chosen template image.

1.3 Image Processing Folders

- `games_images_folder`: Folder containing images for the four games.
- `aligned_images_folder`: Folder where processed aligned images will be stored.
- `cropped_folder`: Folder where processed cropped images will be stored.
- `final_boards_folder`: Folder where final processed boards will be stored.
- `tokens_folder`: Folder containing cropped and processed token images.

1.4 Results Folder

`submission_folder`: Folder where results will be stored. Users may need to adjust this path as well.

1.5 Constants

- `NUM_ROUNDS`: Predefined variable indicating the number of rounds for each game (set to 50).
- `game_numbers`: List of game numbers for the evaluation part, which may need to be adjusted.

2 Align Images

2.1 Function to Retrieve Images for a Game

- `get_images_for_game(game_number, directory)`: Retrieves paths of all images for a given game number from the specified directory (the folder which contains the images for the games).

2.2 Image Preprocessing Functions

- `preprocess_image(image_path)`: Enhances the input image using Gaussian blurring and Contrast Limited Adaptive Histogram Equalization (CLAHE).
- `binarize_image(image)`: Binarizes the input image using Otsu's thresholding method.
- `remove_noise(image)`: Removes noise from the input image using thresholding.
- `blur_image(image, blur_amount=5)`: Applies Gaussian blurring to the input image.

2.3 Function to Load Images from a Folder

- `load_images_from_folder(folder)`: Loads images from the specified folder and returns a dictionary with filenames as keys and grayscale images as values.

2.4 Class for Image Alignment Using SIFT

- `GameBoard(template_image)`: Class constructor that initializes a GameBoard object with a template image for alignment.
- `_get_keypoints_and_features(image) -> tuple`: It is a private method that extracts keypoints and features from an input image using the SIFT algorithm.
- `_generate_homography(all_matches, keypoints_source, keypoints_dest, ratio=0.75, ransac_rep=4.0)`: also a private method that generates a homography matrix based on matched features between source and destination images.
- `_match_features(features_source, features_dest)`: an additional private method that matches features between source and destination images.
- `align_and_save(source_image, output_path)`: Aligns the source image to the template image and saves the aligned image to the specified output path.
- `scale_image_to_template(image)`: Scales the input image to match the dimensions of the template image based on feature matching and homography estimation.

2.5 Image Preparation

- `game_images_dict`: Dictionary to store image paths for each game, where the game number serves as the key.
- Loop: Iterates through each game number to retrieve and preprocess images using `get_images_for_game` and `preprocess_image`.

2.6 Alignment Process

- `initial_template`: Preprocesses the template image for alignment.
- `game_board_template`: Initializes a GameBoard object with the preprocessed template image.
- `aligned_images_folder`: Creates a folder to store aligned images.
- Loop: Aligns each game's images to the template using the GameBoard object and saves the aligned images.

2.7 Alignment example for an image with perspective view

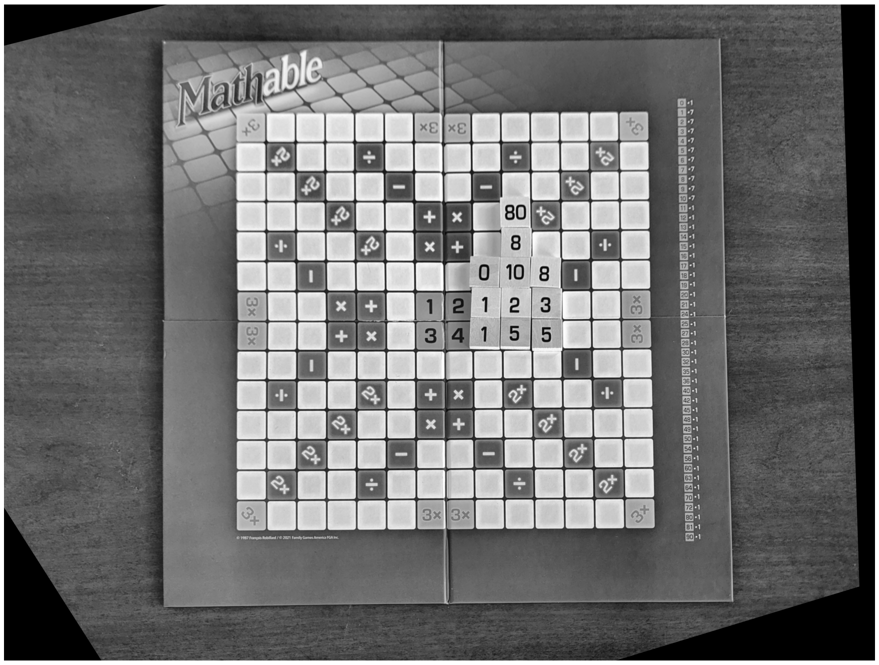

Figure 1: Alignment example for image 3_11jpg, having a perspective view

3 Cropping the Images

I have developed a custom piece of code to manually select the corners of the grid on the board. The reason I opted not to automate this part is because I didn't want to constrain the selection to a perfect square contour. Instead, I aimed for a more flexible approach that would allow for slightly more accessible margins. The best configuration was found through trial and error.

First Approach Explanation

- `clicked_points = []` and `display_image = initial_template.copy()`: Initialize variables for storing clicked points and a copy of the template image.
- `corner_coordinates = []`: Initialize a list to store corner coordinates.

- `select_corners(event, x, y, flags, param)`: Define a callback function to select corners using mouse clicks.
- `cv2.namedWindow('Select Corners', cv2.WINDOW_NORMAL)` and `cv2.setMouseCallback('Select Corners', select_corners)`: Create a window and set mouse callback function to select corners.
- `cv2.imshow('Select Corners', display_image)`: Display the initial template image to select corners.

Then we wait for user interaction to select corners and display selected corners on the image. For simplicity and for accuracy reasons, I opted to hardcode the coordinates of the corners list:

`corner_coordinates = [(1075, 516), (3017, 510), (3052, 2455), (1075, 2468)]`

This configuration will be used for all existing images.

The next section of the code focuses on cropping the images to the grid on the board. It involves defining the corners of the output image, computing a perspective transformation matrix, and applying the transformation to the initial template and all other images.

Main Cropping Code Explanation

- **Defining Output Image Dimensions**: The width and height of the output image with grid cells are defined.
- **Defining Output Corners**: The corners of the output image are defined based on the specified dimensions.
- **Computing Transformation Matrix**: A perspective transformation matrix is computed using the corner coordinates and desired output corners.
- **Applying Transformation to Template**: The perspective transformation is applied to the initial template to crop it.
- **Cropping All Images**: Each image in the aligned images folder is cropped using the same transformation matrix.
  - The base image is loaded and processed first, followed by all other images.
  - The processed images are saved in the cropped folder.

Identifying Rows and Columns of the Board

Code Explanation

- `row_starts = []` and `column_starts = []`: Initialize lists to store the starting points of rows and columns.
- `mouse_callback(event, x, y, flags, param)`: Define a callback function to record mouse clicks for selecting row and column start points.
- `cv2.namedWindow("Select Rows")` and `cv2.namedWindow("Select columns")`: Create windows to display the base image for selecting row and column start points.

- Loop: Display the base image and wait for user interaction to select start points for rows and columns. Press 'q' to quit the selection process and 'c' to clear the selected points.
- Store the selected row and column start points.
- `row_starts` and `column_starts`: Store the hardcoded row and column start points obtained from the manual selection process for future reference.

The rows and columns of the grid board are hardcoded to facilitate the processing. They were obtained through manual selection, as automated detection wasn't feasible due to variations in cell dimensions.

3.1 Defining Grid Cells

In this part of the code, a list is created to store the boundaries of grid cells. Each cell is identified through its top-left and bottom-right corners.

- `grid_cells = []`: Initialize a list to store cell boundaries.
- Loop: Iterate through the row and column start points to create cell boundaries.
  - `cell_boundary`: Define the boundary of each cell using the top-left and bottom-right corners.
  - Append the cell boundary to the `grid_cells` list.

Each cell boundary consists of the coordinates of its top-left and bottom-right corners, allowing for precise identification and manipulation of individual cells.

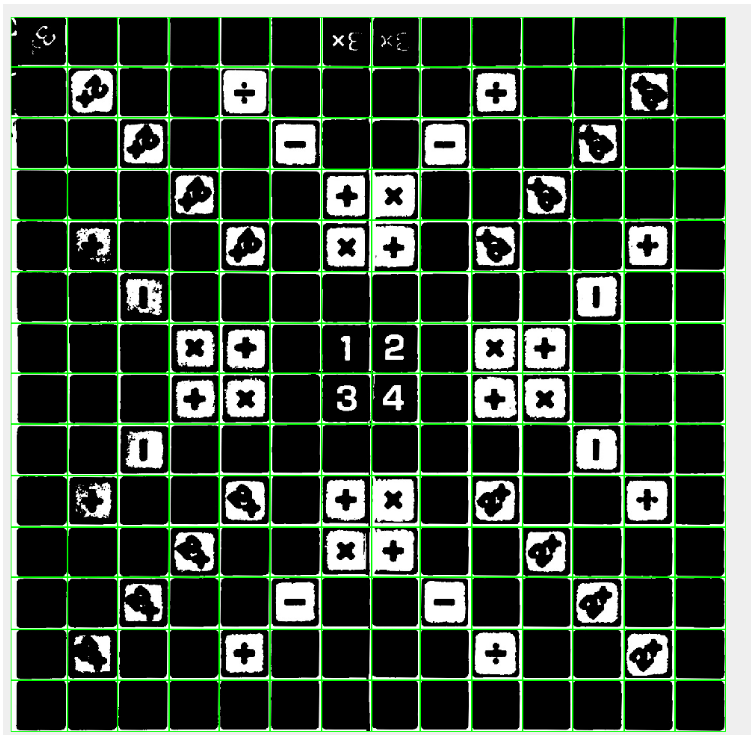

Figure 2: Example of a cropped image with a grid

3.2 Cell Identification Functions

This section defines two functions: one for obtaining the row and column of a cell based on its coordinates, and the other for retrieving the coordinates of a cell based on its row and column.

- `get_row_and_column(x, y, grid_cells)`: This function calculates the row and column of a cell based on the coordinates of its top-left and bottom-right corners.
  - `cell_width` and `cell_height`: Calculate the width and height of a cell.
  - Determine the column index by dividing the x-coordinate difference by the cell width and adding 1.
  - Determine the row index by dividing the y-coordinate difference by the cell height and adding 1.
  - Convert the column index to a letter representing the column (A-N).
  - Return the row and column as a string (e.g., "1A", "2B").
- `get_grid_cell_coordinates(row, column, grid_cells)`: This function retrieves the coordinates of a cell's top-left and bottom-right corners based on its row and column.
  - Calculate the index of the cell in the `grid_cells` list.
  - Retrieve the cell coordinates using the calculated index.
  - Return the coordinates of the top-left and bottom-right corners.

These functions are essential for mapping between cell coordinates and cell indices, facilitating subsequent processing steps.

4 Extracting Tokens

This section focuses on extracting images of all possible tokens from an aligned and cropped image containing them. The extracted tokens will be used for template matching in the next steps.

- `tokens_image`: Load and preprocess the image containing all possible tokens.
- `aligned_tokens_path`: Align and save the tokens image to ensure proper alignment for extraction.
- `tokens_aligned_image`: Load the aligned tokens image and further process it to remove noise and blur.
- `extract_tokens(board_image, grid_cells, output_folder)`: Function to extract tokens based on the grid cells and save them in the specified output folder.
  - `tokens`: List of possible tokens.
  - Loop through grid cells: Iterate over the grid cells to extract tokens.
  - `get_grid_cell_coordinates(row, column, grid_cells)`: Get the coordinates of the grid cell.
  - Extract token: Extract the token from each grid cell by thresholding, finding the bounding box, and cropping the cell.
  - Save token image: Save the extracted token as a separate image in the output folder.

This process ensures that all possible tokens are extracted and saved as individual images, ready for further analysis and template matching.

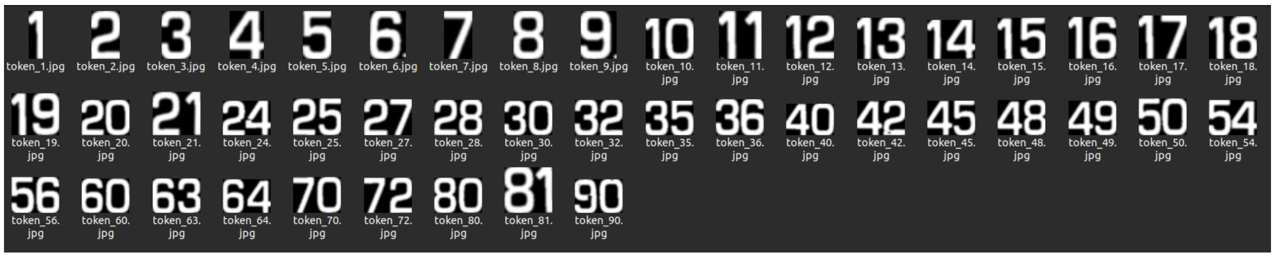

Figure 3: Tokens folder

5 Final steps preparation

5.1 Inverting Center Cells

We begin by inverting the center cells of the grid images to ensure accurate template matching. The `invert_center_cells` function takes an image and the grid cell coordinates as input. It iterates over the coordinates of the center cells, retrieves the corresponding cell regions from the image, and performs bitwise inversion on each cell using the `cv2.bitwise_not` function. The resulting modified image with inverted center cells is stored.

5.2 Processing All Cropped Images

Next, we process all cropped images from a specified folder. For each image, excluding certain ones, we apply the `invert_center_cells` function to invert the center cells and then blur the image. The processed images are saved in a designated folder.

5.3 Loading Tile Images

We load images of all possible tokens from a folder. These images will be used in the template matching part.

5.4 Updating Board Image for Each Round

The `update_board_image` function retrieves the image of the board for a specific round of a game. It constructs the path to the board image based on the round number and game number and returns the corresponding image if it exists.

5.5 Marking Visited Cells

The `mark_visited_cells` function marks the cells on the board image that have already been visited. It takes the original image, a list of visited cells, and grid cell coordinates as input. For each visited cell, it calculates the coordinates of the corresponding grid cell and draws a filled rectangle over that cell on the image to mark it as visited. The function returns the modified image with visited cells marked. It will be used when we'll update the board for template matching (otherwise we risk that a token will be found at the same position twice and we want to avoid this, as it produces unwanted results, as it happend for example for token 10 in game 1).

6 Template Matching

The `main()` function performs template matching to identify and extract the tiles having the biggest scores from a series of game boards. I will explain the logic behind my implementation of the template matching algorithm.

The function begins by setting a threshold value (chosen through trial and error and by carefully examining what score does each token obtain and at what position) for template matching and defining the number of grid rows and columns (which is a constant: the board has 14 rows and 14 columns). It also includes a helper function `get_neighbors()` to obtain neighboring cells for a given cell on the game board. The reason for which this function is implemented is quite simple: I wanted to search in the vecinity area of previously placed tiles, because the rules of the Mathable game do not allow us to place a tile anywhere on the table. With this function, I also improve the complexity of the algorithm and I assure a higher accuracy for my predictions (for example I observed that a token with the digit 1 could be associated to a grid line if we are not careful enough).

For each game round (from 1 to 50) and game number (ranging from 1 to 4), the function iterates over the visited cells and retrieves the corresponding board image. It then marks the visited cells on the board image to avoid redundant matching.

The function proceeds to sort and iterate over the token images in descending order of token numbers. It prioritizes double-digit tokens and rotates them at various angles for better matching accuracy. Using template matching with normalized cross-correlation (`cv2.TM_CCORR_NORMED`), the function compares each token image with the modified board image.

If a high-scoring tile is found among the double-digit tokens (based on the specified threshold), it records the tile's position and token number. If no high-scoring tile is found among double-digit tokens, the function repeats the process for single-digit tokens.

Finally, the function selects the highest-scoring tile from the identified high-scoring tiles and records its position and token number. The results are saved in text files named according to the game number and round number.

The reason for which we check the double-digit numbers first is the following: I observed that, for number 23 for example, a really high score will be obtained for digits 2 and 3 (a score very close to the threshold). Therefore, there were some cases in which my algorithm would predict wrong, identifying a double digit number with a single digit number. To avoid this behaviour, I made the algorithm look through double digits number first, and if there were ones that would have a score exceeding the threshold, then I will store them in a high scoring tiles list. Why wouldn't I take the first match? Let me offer you another example: for tokens 49 and 48 we usually obtain very similar scores (same thing happens for 18 & 19, 80 & 60 and a few other pairs of numbers). Usually, the score for the 49 is very close to the threshold, but the score of 48 would be 0.99/1. Therefore, the best match would be found between the tokens that have a score greater than the threshold, but also the highest score from that list.

7 Creating the Mathable Board

I will present the process of creating a mathable board, which involves initializing a grid with special places and operands.

- `create_mathable_board()` Initializes an empty 14x14 board and places special places and operands on it and returns a 2D list representing the mathable board with special places and operands.

In the function `create_mathable_board()` we define special places such as double and triple score cells, along with mathematical operands like addition, subtraction, multiplication, and division signs. Additionally, it includes center tokens on the board for further evaluation. These all will help us follow correctly the rules of the game to compute the score.

8 Scoring process

8.1 Helper Functions for Scoring

This section presents two essential helper functions used in the scoring process, which involve reading and organizing data related to game rounds and player turns.

- `read_round_info(results_folder)` Reads information about game rounds from text files in a specified folder and stores the data in a dictionary (with keys representing game numbers, and values some lists containing tuples of the following type: ('1A', 8)).
- `read_player_turns(folder)` Reads player turns data from text files in a specified folder and organizes the data into a dictionary (with keys representing game numbers, and values some lists containing tuples of the following type: ('Player 1', 2)).

8.2 Evaluating Expressions

- `evaluate_expression(board, row1, col1, operand, row2, col2)` It computes the result of an arithmetic expression using the specified operands and cell coordinates on the mathable board. This helps us check if a token can be obtained from multiple equations (for different adjacent token pairs).

8.3 Computing Round Scores

I will outline the process of computing scores for each round based on the actions performed on the mathable board.

- `compute_round_score(old_board, new_board, cell, token)` Computes the score for a single round based on the actions performed on the mathable board (token placement and mathematical operations performed on the board).
- `compute_player_score(initial_board, positions_and_tokens, player_turns, output_folder)` Computes the score of a player for each turn across all rounds in a game and stores the results in text files.

This process ensures that the score for each player's turn in every round is accurately computed and recorded for further evaluation.

The algorithm ends at this point.