Project documentation - Mathable

Ciuperceanu Vlad-Mihai

1 Introduction

This project aims to analyze some Mathable games, extracting information from the images provided after each move and solving the 3 given tasks. The first task is related to finding the position where the piece was placed, the second is related to identifying the placed piece, and the third task is related to calculating the score for each player. Among these, the first two require Computer Vision techniques, while the third task only uses the information obtained previously.

Overall, the steps we go through to solve the requirements are:

- extracting the large board (removing the table from the images)
- extracting the small board (removing the border of the large board)
- dividing the obtained board into a 14x14 grid, obtaining the squares where the pieces will be placed
- traversing the images and finding the difference between the current image and the previous one (and finding the position where the last piece was placed)
- identifying the piece that was placed using template matching
- retaining the score obtained for this move and, subsequently, determining the score for the players for each of their rounds

The first 4 steps are related to the first task, while step 5 corresponds to task 2, and the last step is related to task 3.

In the following, we will present the methods by which we approached these steps, solving the 3 tasks.

2 Precalculations and observations

First of all, we can analyze the images before processing them, extracting some general information about the given images.

We can initialize some constants with data from the game and the images. For example, we can note that the large board has dimensions of 1975x1975 pixels, that the margin between the large board and the small board is 250 pixels, and that the small board has dimensions of 1456x1456 pixels. These values are approximate, being observed from analyzing an image opened with OpenCV. Due to subsequent operations, these numbers do not need to be fixed, only close enough to help us in the process of analyzing the images. Also, the number 1456 was chosen because it is divisible by 14, the number of squares we have in our grid, leaving 104 pixels per square (both in width and height).

Information about the squares can be stored using a class, containing all the information needed for the tasks:

- the position
- the value of the placed piece (if any)
- the bonus on that square (1 if it is not a special square)
- the condition that must be respected in creating the equations

Also, since the board is standard, we can initialize a 14x14 matrix that represents the information from an empty board, where no piece has been placed yet. The bonus and conditions are static, so they can be stored from the start. The values of the pieces will also be stored from the beginning, to be used in task 2.

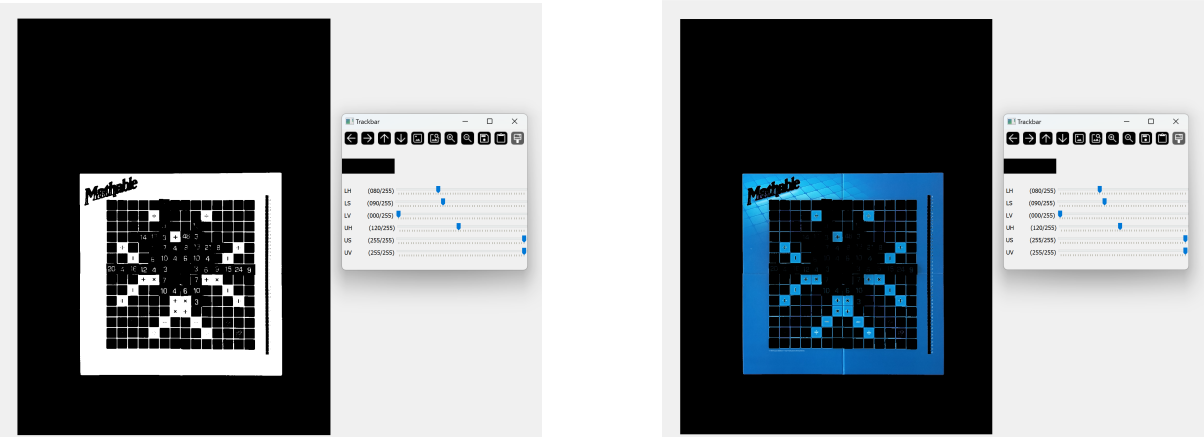
We also notice that, for the small board, we find the top-left corner at coordinates (257, 257), and the bottom-right corner at coordinates (1713, 1713) - relative to the corners of the large board. These will be useful later, in extracting the small board.

3 Task 1 - Finding the position where the piece was added

As mentioned earlier, there are several steps we need to go through to obtain the information from each image, then compare the obtained information to determine the position of the new piece.

3.1 Extracting the large board

For this step, the first thing to do is to remove the table from the image, then take the contour of the large board. Thus, we will use the HSV representation of the image to generate a mask and transform the given image, highlighting the large board. After several attempts, using the code from the lab, we arrived at a set of values that facilitate this operation: $LH = 80$, $LS = 90$, $LV = 0$; $UH = 120$, $US = 255$, $UV = 255$. Below are 2 illustrations showing the effect of this transformation on the image $1\_41.jpg$ from the $fake\_test$ dataset:



(a) Mask (b) Res

Figura 1: Transformations using HSV representation

Next, we will apply sharpening to the obtained image, to better highlight the contour of the large board, followed by thresholding to obtain a binary image and erosion to remove noise, similar to lab 6.

We will apply Canny Edge Detection and the $findContours$ function from OpenCV to obtain the contour of the large board. By traversing it, we will determine the corners of the large board, to be able to extract the entire large board from the image. The corners are obtained by searching for points whose coordinates' sum or difference is minimum or maximum, respectively. We obtain the result in Figure 2 - (a). If we apply the $getPerspectiveTransform$ and $warpPerspective$ functions from OpenCV, we can obtain the image without the table, as seen in Figure 2 - (b).

(a) Detecting the corners

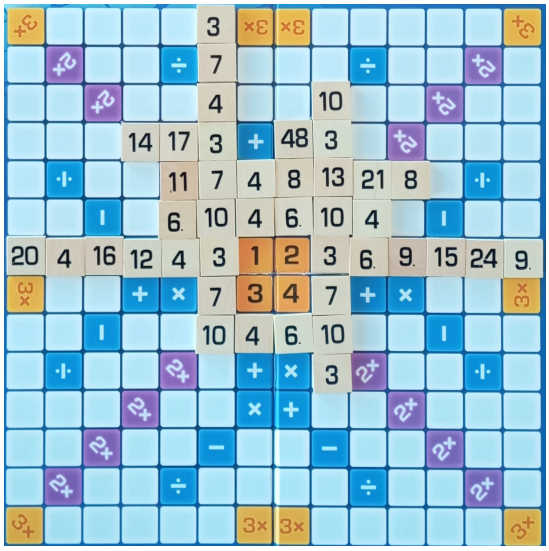
(b) Extracting the large board

Figura 2: Detecting the corners and extracting the large board

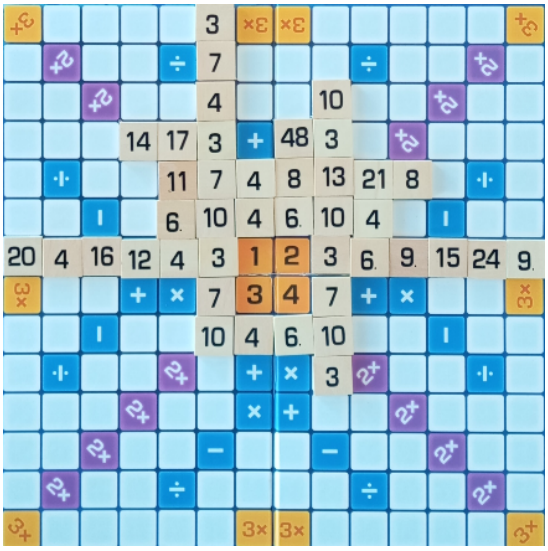
3.2 Extracting the small board

To extract the small board from the large one, we can have 2 methods: either extract the board directly manually, using the approximate coordinates of the corners (with small variations), or use an approach similar to the above, with the 2 functions from OpenCV. The first option works quite well, but unwanted pixels may appear at the edges due to the approximation error of the coordinates, as well as the fact that the small board is not perfectly straight in the image. The difference between the 2 approaches can be seen in the figure below, justifying the choice of the second method.

Thus, we have delimited the region of maximum interest from the given images, to extract the necessary information. From this point on, we will use this image processing for the following steps, having all the data about the move in this image.



(a) Manual crop



(b) Using OpenCV functions

Figura 3: Extracting the small board

3.3 Dividing the small board into grid squares

The grid delimitation will be done manually, constructing the lines that will delimit the squares at a fixed distance of 104 pixels, as we calculated earlier about the small board.

This delimitation will help us identify patches on the board, tracking the places where the pieces will be placed.

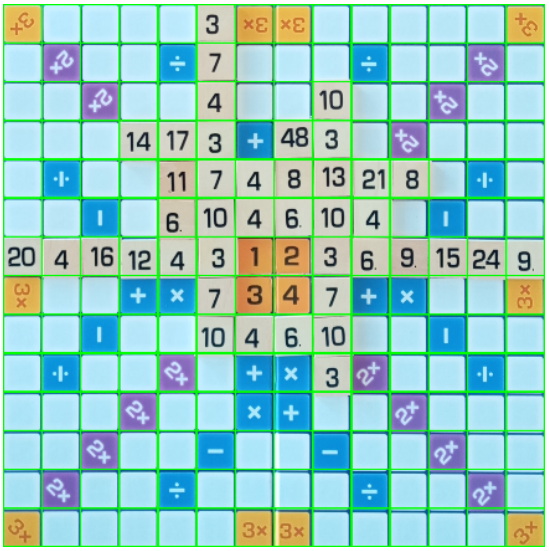


Figura 4: Delimited board

3.4 Finding the position of the added piece

Now that we have extracted the small board, we have reached the stage where we need to determine the position of the added piece.

For this, we will traverse the images with each move of a game, and we will be interested in the difference between the current image and the previous one. Obviously,

the first image of a game will be compared with the image of the empty board, extracted from the auxiliary image 01.*jpg*. This difference will give the position of the new added piece. Thus, we transform both images into grayscale, apply a threshold, and calculate the absolute difference between them. By going through the patches delimited earlier (which correspond to the squares on the board), we will look for the maximum difference between the 2 images, to determine the position of the added piece.

Of course, there will be differences in other places than where the piece was placed, due to changes in the brightness of the environment or the shadows created by adding a piece. Therefore, we will cut a little from the edge of the patch, looking more at the center of it, where the changes will be more significant and where shadows or other changes should not be felt as much. Also, we will check that we do not already have a piece placed in that square from a previous image, to exclude differences that are not caused by adding a piece.

Thus, we will find the position of the piece that was added in the current image.

4 Task 2 - Identifying the added piece

After finding the patch where a new piece was added, it remains to identify it using template matching.

4.1 Obtaining the templates

First, we need to obtain a database with the templates of the pieces, to compare them with the patch found earlier.

This is found in the *templates* folder, where we have a template for each piece. They were obtained by manually extracting each piece in images of 103x103 pixels, using the auxiliary image 04.*jpg*, where there is space between the pieces, ensuring good separation and positioning of them.

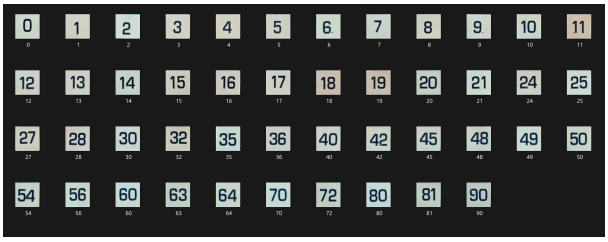


Figura 5: Templates of the pieces

We tried to extract them as close as possible to the contour of the piece, with the number as centered as possible, to facilitate template matching, but, obviously, human error can be observed in this operation. Brightness and shadows are also another factor in this operation. Therefore, when performing template matching, we will cut a little from the edges of the templates, to avoid edges that can bring unwanted pixels that contribute to the correlation level.

4.2 Template matching

After obtaining the patch where the piece was added, we will send the respective patch to a function that will perform template matching with the templates obtained earlier.

Before applying the *matchTemplate* function from OpenCV, we will perform some transformations on the patch and the templates.

First, both will be transformed into grayscale, and thresholding will be applied to obtain a binary image. The imposed threshold is 65, obtained after several attempts on the test data. It is interesting to note that, without this threshold, template matching worked quite well, having only 5 images where it returned a wrong value. Choosing a wrong threshold could lead to possibly worse results. We also tried using the Otsu method, to automatically find the threshold, working almost perfectly, with only one image (1_42.*jpg* from the training set) where template matching failed.

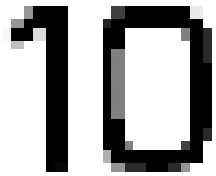


Figura 6: Template for piece 10 after the described operations

Second, the patch sent to template matching will be slightly cut in the edge portion, for the same reason, to not bring unwanted pixels that influence the correlation. Also, we will enlarge the patch with a white pixel border, using the *copyMakeBorder* function from OpenCV. This is extremely important since the templates are slid over the given patch, so sufficient size is needed. Also, the templates may contain the numbers slightly offset from the center of the square, so sufficient margin is needed to find them and make the match, thus leaving more room for positioning.

Finally, we will use the *matchTemplate* function and find the maximum similarity obtained with one of the templates to identify the correct number of the piece. It is worth noting that the checks made at this point, as in the case of identifying the position of the new piece, are sensitive to previous results, influencing each other and potentially propagating errors further.

5 Task 3 - Calculating the score

For this part, we will retain the score brought by adding each piece within a move of a game, to later calculate the scores.

The equation checks are made according to the described rules of the game, retaining, in a matrix, each score brought by a move within a game. We will be careful to check all possible equations obtained from the 4 directions, also considering the conditioning or bonus of the position we are on.

Then, by traversing the files with the players' rounds, we can determine the scores obtained by summing the elements from the respective matrix for a range of moves. At this point, retaining all the board information in the matrix becomes useful.

Finally, we write the obtained results in the score files, along with the identified moves.