UNIVERSITÀ DI TRENTO
Department of Information Engineering and Computer Science

Artificial Intelligent Systems - Signal Image and Video

PROJECT REPORT

Rubik's cube scanner and solver

developed by Stefano Bonetto

*U*niversity of Trento
Department of Information Engineering and Computer Science
Via Sommarive 8, 80123
Povo (TN), Italy
stefano.bonetto@studenti.unitn.it

Abstract

This project tackles the task of solving a Rubik's Cube without resorting to machine learning. Instead, it relies on straightforward image processing techniques combined with the Kociemba algorithm, presenting a streamlined and efficient solution to conquer the puzzle. My primary goal was to develop a solver that is very fast both in the scanning phase and in finding the solution. To achieve this, I made certain trade-offs and simplification assumptions.

The complete code is available on my Github.

Contents

1 Introduction

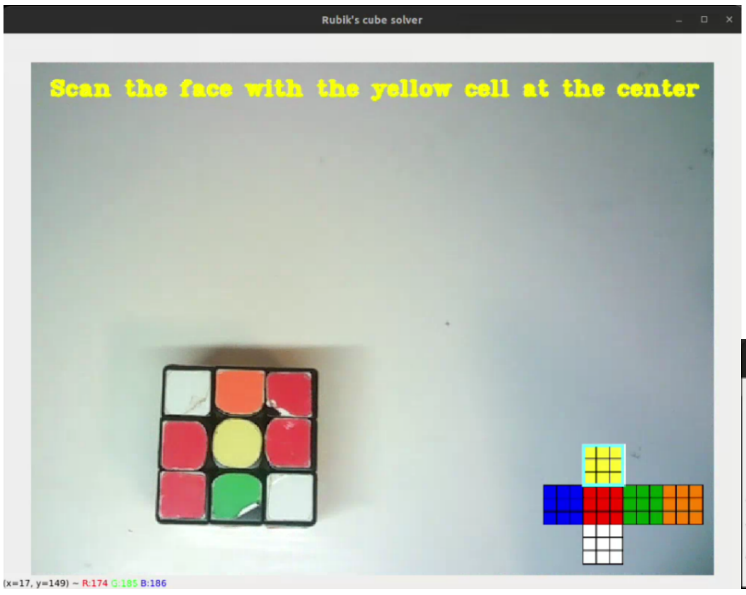
The application is structured around two core phases:

- **Scan of the Cube's Faces:** this involves the comprehensive examination of each face of the Rubik's Cube, employing color detection to discern the composition of every face.
- **Formulation of the Cube's Solution:** `kociemba` algorithm is applied in this phase to derive the optimal solution for the Rubik's Cube. Additionally, an intuitive Graphical User Interface (GUI) is provided to guide the user through the solving process.

2 Scan of Cube's Faces

The initialization of the application starts with the projection of two windows (as shown in the Figure 1):

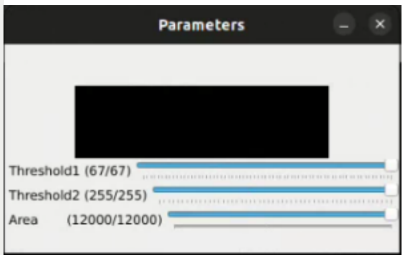
- The initial screen, labeled as "*Rubik's Cube Solver*," offers users a live webcam feed, complete with prompts indicating the specific face they should scan at that moment.
- The "*Parameter*" window helps us to choose the thresholds for the range of the Canny algorithm and a tollerance



Scan the face with the yellow cell at the center

(x=17, y=149) ~ R:174 G:185 B:186

(a) *Webcam feed window.*



Threshold1 (67/67)
Threshold2 (255/255)
Area (12000/12000)

(b) *Parameters' window.*

Figure 1: *Initial screen.*

2.1 Pre-processing phase

The first processing operations prepares the frames for the color detection phase.

First it applies a Gaussian blur to the image to reduce noise and details in the image, resulting in a smoother appearance. Then it convert the image in Gray scale.

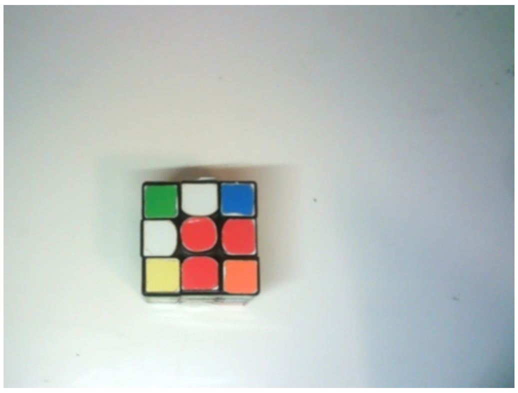
To detect face's edges I choose the **Canny edge detection algorithm** which is one of the most popular edge detection algorithm in image processing. It works by detecting areas of the image where the intensity changes sharply, these areas are typically interpreted as edges. The algorithm uses two threshold values that can be updated in the *Parameters* window:

- `threshold1`: this is the lower threshold. Any gradient value lower than this is considered not an edge.

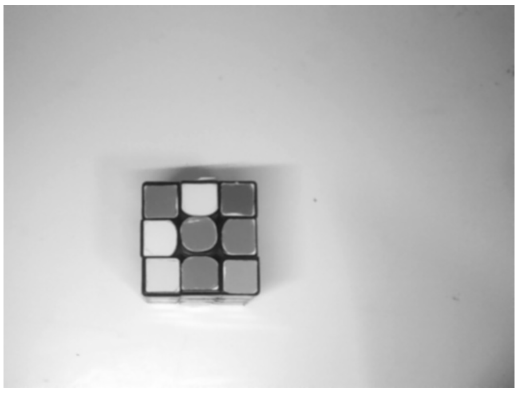
- `threshold2`: this is the higher threshold. Any gradient value higher than this is considered an edge.

By changing these values we basically choose what constitutes an edge.

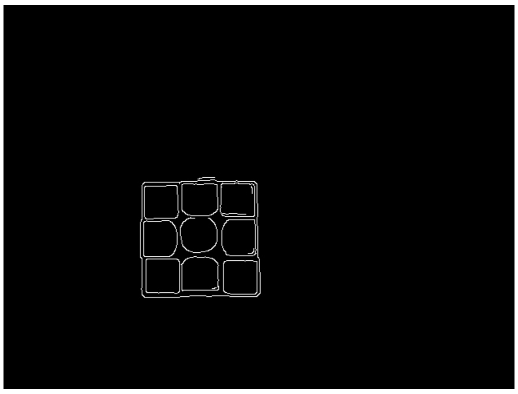
Finally I dilate the edges detected by the Canny operation, this is useful because it makes the edges more pronounced.



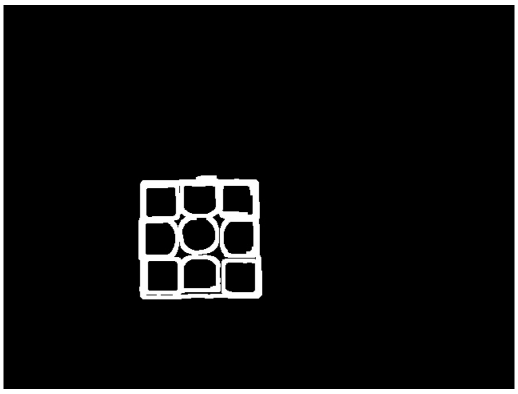
(a) *Blurred image*



(b) *Gray-scale image*



(c) *Canny algorithm applied*



(d) *Edges dilated*

Figure 2: *Pre-processing steps*

2.2 ROI cropping

Using the `getContours` function, I analyze the pre-processed image and identify contours that surpass a designated threshold, referred to as `minArea`, a parameter adjustable in the *Parameters* window. For instance, in Figure 2d, the algorithm considers only the outer contour outlining the entire face, disregarding smaller contours corresponding to individual squares (those with an area less than `minArea`).

At the conclusion of the `getContours` function, the original image is cropped, preserving solely the Region of Interest (ROI) associated with the face, as depicted in Figure 3. Six ROI's images are then stored in the `faces` folder.

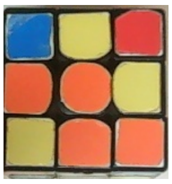


Figure 3: *Cropped ROI*

3 Cropping of single squares

To enhance efficiency in distinguishing the nine squares within each Region of Interest (ROI) on a face, I initially considered reapplying the `process_frame` and `getContours` functions, utilizing a threshold (`maxArea`) to retain only the edges outlining the squares. However, due to the computational intensity of this approach, I opted for a more lightweight alternative.

The revised strategy involves subdividing the ROI obtained in previous steps into nine equal zones, achieved by dividing both the height and width of the image by three. Subsequently, I identify the center of each resulting square and extract a smaller square, reduced in size by a factor of six and centered at the same coordinates. This has been doing to achieve problems due to bad crop operations in the precious phases. This streamlined approach balances efficiency with a reduction in computational complexity, however it's not as robust as the previous approach.

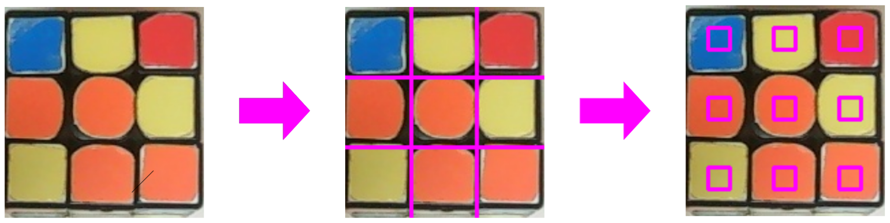


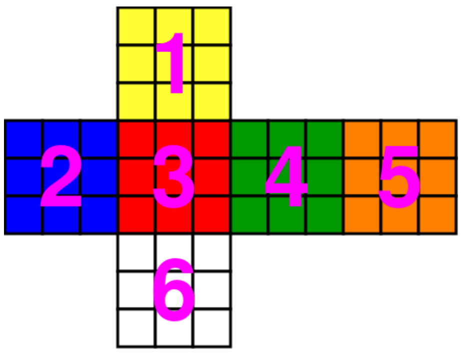
Figure 4: *Crop steps*

The image is illustrative. In the final step, larger squares are displayed compared to the ones scanned. This is because showing the actual squares would result in them being too small ($\frac{1}{18}$ of the edge of the face).

For each face, the process yields nine distinct images, each of which is then saved in the `squares` folder.

4 Detection of reference squares

I've asked the user to follow a specific order when scanning the face initially:



In this phase, I utilize this knowledge as a reference for color detection. For instance, I know that the central cell (square number 5) of the first face is expected to be yellow. Taking into account the prevailing illumination conditions, I fine-tune the range of yellowish colors by calculating the mean values in the HSV color space for the 5^{th} square of the first face. Subsequently, I establish both a lower bound and an upper bound for the hue (h) with a margin of ± 10. I repeat this operation for all six faces.

The final outcome is a dictionary that includes both upper and lower bounds for each color.

5 Color detection and cube virtual reconstruction

The subsequent step is straightforward: I iterate over the 6 faces and individual squares, calculating the mean of the HSV values and identifying the color based on the predefined ranges. Special cases are handled as follows:

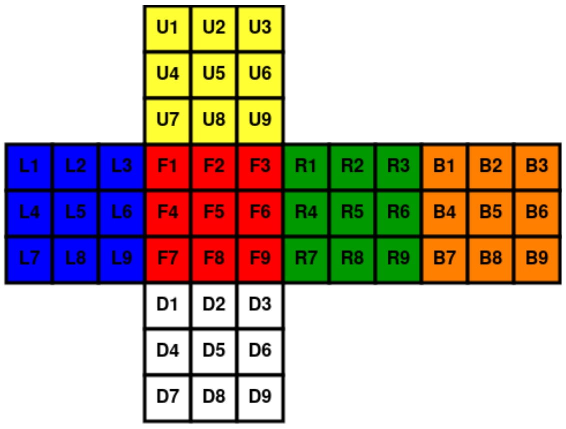
- If the mean HSV values do not fall within any of the specified ranges, I assign that square to the nearest color value[1];
- In the event that a square is classified as a red or orange cell, I perform an additional verification, as these colors are closely related: the cell is then assigned to the nearest value[2].

Ultimately, we obtain a reconstructed representation of the cube, which is projected onto the terminal.

6 Solution

I proceed to compute the solution of the cube using the `kociemba` module.

The `kociemba.solve(string)` function requires a string parameter representing the cube, which is in a different notation than the one employed in the remainder of my program:


U1 U2 U3
U4 U5 U6
U7 U8 U9
L1 L2 L3 F1 F2 F3 R1 R2 R3 B1 B2 B3
L4 L5 L6 F4 F5 F6 R4 R5 R6 B4 B5 B6
L7 L8 L9 F7 F8 F9 R7 R8 R9 B7 B8 B9
D1 D2 D3
D4 D5 D6
D7 D8 D9

The `kociemba` algorithm accepts strings with the following notation order: `U1, U2, U3, U4, U5, U6, U7, U8, U9, R1, R2, R3, R4, R5, R6, R7, R8, R9, F1, F2, F3, F4, F5, F6, F7, F8, F9, D1, D2, D3, D4, D5, D6, D7, D8, D9, L1, L2, L3, L4, L5, L6, L7, L8, L9, B1, B2, B3, B4, B5, B6, B7, B8, B9`.

To accommodate this requirement, I have implemented a conversion algorithm that allows me to adapt the Rubik's Cube notation.

The `kociemba.solve(string)` function returns a list of moves to execute in order to solve the Rubik's Cube, written in the correct notation. This assumes that you're looking at the cube with the front face having the red central cell.

We have six possible layers to move:

- Right layer (`R`)
- Left layer (`L`)
- Front layer (`F`)

[1]In the detection of reference square phase, I saved the hue value to compare it in this phase.
[2]This is one of the major issues of this algorithm. Sometimes, due to the low resolution of the webcam, this similarity is imperceptible even to the human eye. It is definitely the Achilles' heel of this approach.

- Back layer (`B`)
- Upper layer (`U`)
- Lower layer (`D`)

Each move can be 90 degrees clockwise (e.g., `R`) or 90 degrees counter-clockwise (e.g., `R’`). If a 2 is appended to the move, it signifies a 180 degrees rotation of the layer (regardless of whether it is clockwise or counter-clockwise, as they are equivalent).

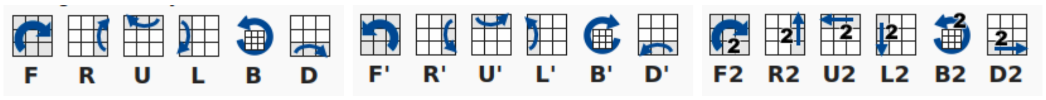


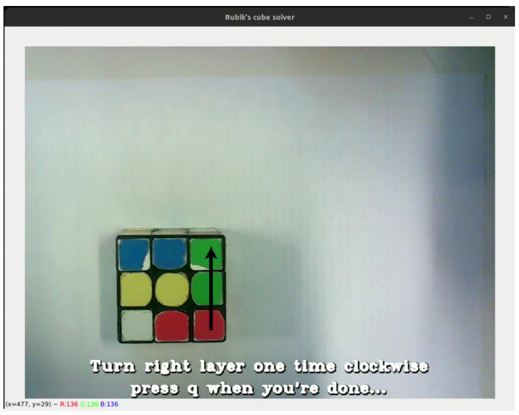
Figure 5: *All the possible moves used to solve the cube, images are taken from https://ruwix.com/the-rubiks-cube/notation/.*

7 Solution, graphical implementation

Once I’ve obtained the list containing the moves to solve the cube, I want to add some visual tips to help the user to solve the cube.

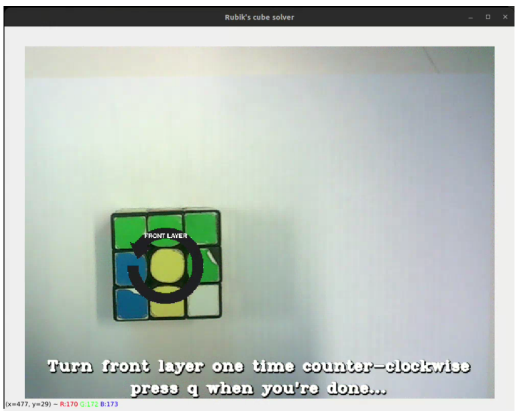
I attach arrows to the cube, as demonstrated in Figure 5, to monitor its movements. I consistently employ `process_frame` and `getContours` functions, running them continuously to adapt to any potential adjustments in the cube’s orientation due to movements by the user.

While this process demands more resources compared to, for example, tracking the cube just once and attaching the arrow solely in that position, it significantly enhances the flexibility of the tracking mechanism. Infact that last approach will be significantly less expensive, but it’s inconsistent and useless if we don’t assume that the cube remains stationary during the interim period.



Turn right layer one time clockwise
press q when you're done...

(a) *R move*



Turn front layer one time counter-clockwise
press q when you're done...

(b) *F’ move*

Figure 6: *Some tips given by the program to the user to help him solving the cube.*

An interesting process is the one used to overlap the arrow corrispondent to the move we have to perform over the webcam feed. Once we’ve obtained the coordinates of the cube from the `getContours` function, we start by reading the arrow (png) image. We resize it according to his position then we create an inverse binary mask of the image, where pixels where the arrow is present are 0 (black) and the ones where the arrow is not present are 255 (white).

When we apply this mask to the ROI, basically we’re erasing the area in the ROI where the arrow will be placed (we obtain `roi_with_arrow`), then we crop just the arrow from the original png image (obtaining `arrow_with_roi`) and we combine them to obtain the final result.

8 Conclusions and future improvements

In conclusion, this project successfully tackles the Rubik's Cube-solving challenge through a combination of image processing techniques and the `kociemba` algorithm.

The scanning phase involves some pre-processing steps, including Gaussian blurring, Canny edge detection, and dilation, to enhance the accuracy of face detection. Subsequent steps, such as Region of Interest (ROI) cropping and color detection, contribute to the virtual reconstruction of the cube. Reference squares aid in the identification of colors. Furthermore, the implementation includes a graphical representation of the solution, integrating arrows to guide users through the solving process. Although this method demands more resources due to continuous tracking, it offers enhanced flexibility, providing real-time visual assistance.

However, there exist potential improvements in the current implementation. One crucial aspect is the optimization of the scanning phase to not only reduce computational intensity but also enhance the program's flexibility concerning variations in illumination. The current algorithm has been designed under the assumption of stable illumination conditions, which may limit its performance in diverse lighting environments. Exploring methods to enhance its robustness and adaptability would be beneficial.

Another area for enhancement lies in improving the robustness of the square cropping approach, which directly impacts the accuracy of color detection. An alternative strategy could involve obtaining the edges of individual squares using the Canny edge detection algorithm (as mentioned before). While this approach might offer improved accuracy, it does come with a trade-off, requiring more powerful hardware to handle the increased computational load.

Additionally, experimenting with different tracking mechanisms and evaluating their impact on overall performance could provide valuable insights. Assessing the efficiency and reliability of alternative tracking methods may lead to a more optimized and adaptable system.

In summary, these potential improvements aim to make the algorithm more versatile, robust, and efficient, paving the way for enhanced performance and accuracy in solving the Rubik's Cube across various scenarios and conditions. However, the project successfully achieves its goal of solving the Rubik's Cube without resorting to machine learning. The combination of image processing techniques and algorithmic approaches presents a viable and accessible solution.

9 References

https://github.com/muodov/kociemba

https://ruwix.com/the-rubiks-cube/notation