Rubik's Cube Solver Final Report

Sophie Zheng (sz374), Matthew Xu (mx68)
Github: https://github.com/sophiezheng0711/rubiks-cube-solver

Summary

Rubik's cube solving has been a popular activity amongst children and teens both competitively and casually. However, the algorithm can be easily forgotten, especially after a long period of hiatus. We want to create a computer vision-based Rubik's cube bot to help solve Rubik's cubes. The final product creates a live video stream, where the user scans the faces of unsolved 3 by 3 by 3 Rubik's cube, and the app provides step-by-step instructions to solve it. This project involves two main components: the computer vision component, which involves edge/corner detection, color detection, and reflecting the solution in the video stream; the Rubik's cube solving algorithm, which involves deciding on an algorithm and implementing it. The final version of this Rubik's cube solver uses Kociemba's algorithm, which will be further discussed in this paper. We were able to achieve what we proposed, and our end result can walk a Rubik's cube amateur (such as myself) through solving a cube usually within 20 steps.

Cube Recognition

We decided to use a computer vision-based approach for the cube recognition component of our project. The cube detection problem is essentially a two-phase classification problem, with the first phase being a cube classification problem, and the second a color classification problem. When it comes to classification problems, a deep learning-based solution might come to mind first. However, we know that there are 6 faces on the cube, with 9 squares on each face. This problem is then significantly simplified, and a deep learning-based solution may be unnecessarily complicated. Additionally, a deep-learning approach is not a great method for especially the second phase of our project, since certain colors can look very similar under some lighting conditions (e.g. red and orange). There are many cases where the human eye fails to differentiate between some colors under bad lighting, which will make the deep learning approach more difficult. According to the paper *Color Recognition For Rubik's Cube Robot* that discusses deep learning and cube recognition, the average accuracy of the detector falls between 84% and 88% (Liu et al). Thus, we have concluded that a computer vision-based

approach would be more appropriate for this problem. In this section, we will discuss the design and implementation of the edge detector, color detector, as well as the instruction display for the live video stream. The main packages we will be using here include OpenCV for image processing functions, and NumPy for transformations and matrix calculations.

Preprocessing

Before we start computing the locations of corners and identifying edges, we must preprocess the frame. Although our end goal of this section is to identify the 9 colors of the face, when identifying edges/corners, colors are a major distraction. Thus, the first step is to convert the image to grayscale.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Next, the frame given is generally very noisy: for example, the person holding the cube could be wearing a very distracting shirt, or the wall behind the user may have too many patterns. To avoid identifying edges and corners from the noisy areas of the frame, we must blur the frame. In this case, we used a Gaussian blur with a 3 by 3 kernel. The kernel size determines the magnitude of the blur. If the size is too large, the detector may fail to recognize the necessary edges and corners of the actual Rubik's cube. If the size is too small, the detector may be distracted by the noise in the frame.

```
blurred = cv2.GaussianBlur(gray, (3, 3), 0)
```

At this point, we generally have an image with clear edges, and the cube should be a large, identifiable object in it. Now, we need to detect edges from this processed image. A commonly used edge detector is the Canny edge detector. Its process can be broken down into the following steps: First, apply a Gaussian filter to reduce the noise. Then, find the intensity gradients and directions at every pixel of the image. Next, apply non-maximum suppression, and apply double threshold to find potential edges (hysteresis). The gradient magnitude is calculated by taking the distance between the directional derivatives of the image, which is generally found by convolving the image with an edge detection operator, like the Sobel filter.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

$$G = \sqrt{G_x^2 + G_y^2} \qquad \theta = \arctan \frac{G_y}{G_x}$$

The idea of non-maximum suppression is for each pixel q with image intensity $g(q)$, we take a step forwards and backwards along the gradient: $r = q + \frac{g(q)}{||g(q)||}$, $p = q - \frac{g(q)}{||g(q)||}$. Then, we find the gradient magnitudes at p, q. If the magnitude at q is greater than that at p and r, q is a local peak.

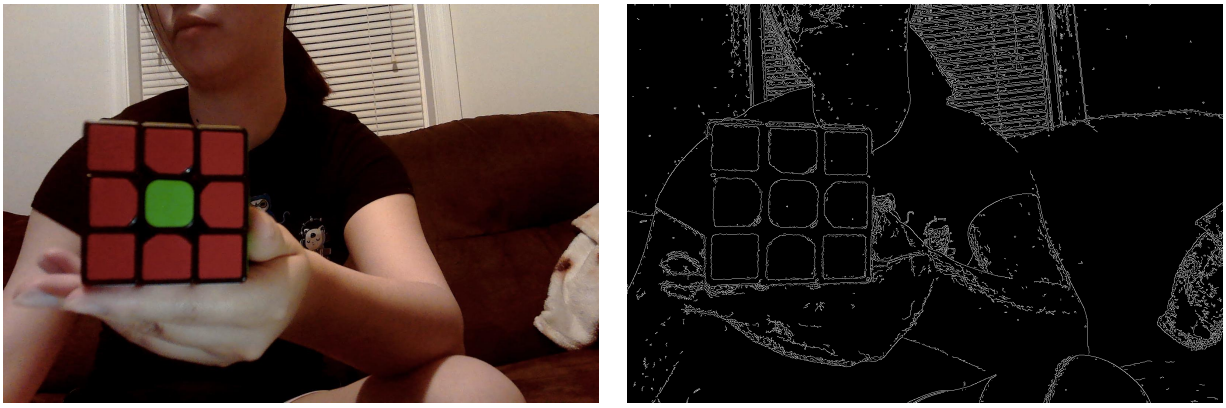


Figure 1: Sample frame vs. frame after Canny edge detection

Finally, we dilate the lines (make them thicker) to prepare for calculating the contours, which will be discussed in the next subsection.

Figure 2: Dilated frame after edge detection

```
# perform edge detection with canny edge detector
canny = cv2.Canny(blurred, 20, 40)
# make lines thicker by dilating
```

```
kernel = np.ones((7, 7), np.uint8)
dilated = cv2.dilate(canny, kernel, iterations=2)
```

Contours

After the image preprocessing discussed in the previous subsection, we need to find the contours in the given frame. According to the OpenCV documentation, contours are curves joining all the continuous points (along the boundary), having the same color or intensity. Contours are useful for shape analysis, as well as object detection, which are fundamental components of this part of our app. The built-in OpenCV function elegantly returns a list of contours in vector form, as well as the hierarchy of the contours. For each contour, the hierarchy contains information including the next contour, the previous contour, the first child contour, and the parent contour. This information will be extremely useful when we prune the overlapping/intersecting boxes.

```
contours, hierarchy = cv2.findContours(
    dilated.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE
)
```

Once we have the contours, we can use OpenCV's approxPolyDP function to adjust the contours into their nearest polygons. This makes pruning non-rectangular contours easier.

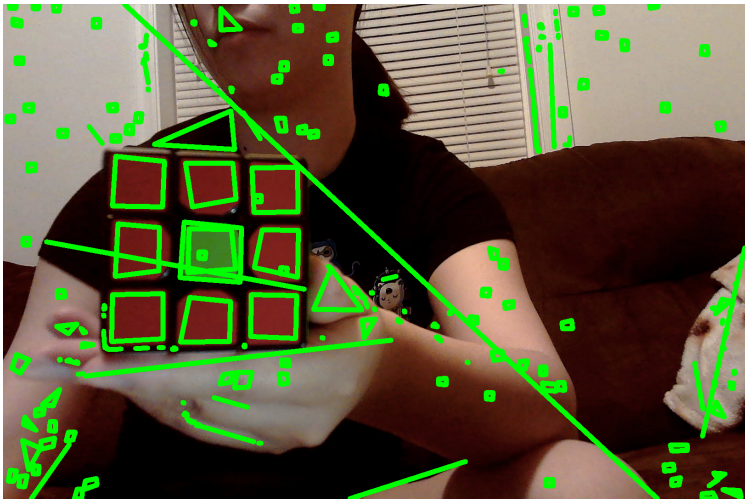


Figure 3: Approximated polygonal contours

To make the computations easier, we package the approximated polygonal coordinates, hierarchy, and hierarchical index of each contour into a tuple list:

```
items = [
   (
       cv2.approxPolyDP(contour, 0.1 * cv2.arcLength(contour, True),
```

```
closed=True),
        rel,
        i,
    )
    for i, (contour, rel) in enumerate(zip(contours, hierarchy[0]))
]
```

Next, we want to prune irrelevant contours, until there are only 9 squares left in the frame. An obvious way to begin this process is to prune all non-rectangular contours. To achieve this, we filtered out contours that do not have 4 sides, ones that do not meet the side length thresholds, and ones that are not convex. For our specific use-case, we determined a threshold of [30, 100] pixels for the side length, which is calculated by taking the square root of the area of the contour. These numbers depend highly on the size of the screen of the app. Checking the number of sides is simply checking the length of the contour, and we used the OpenCV function isContourConvex to determine if a contour is convex:

```
items = list(
    filter(
        lambda item: len(item[0]) == 4
        and np.sqrt(cv2.contourArea(item[0])) > min_side_length_threshold
        and np.sqrt(cv2.contourArea(item[0])) < max_side_length_threshold
        and cv2.isContourConvex(item[0]),
        items,
    )
)
```

Now, we should have a clean image with almost the right contours.

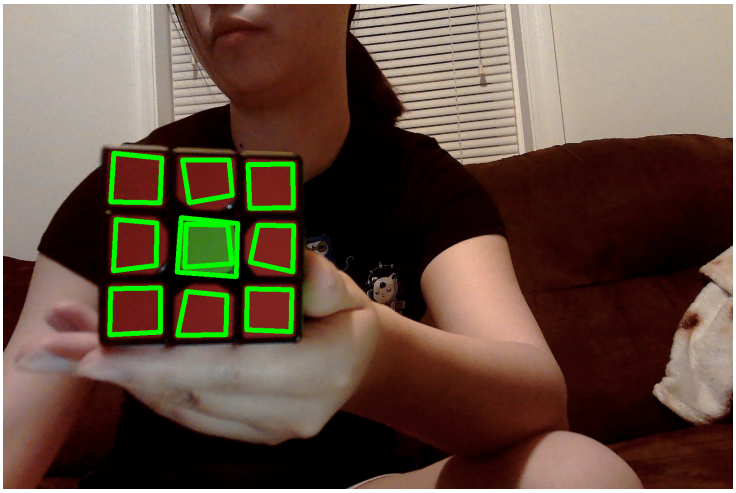


Figure 4: Contours after preliminary pruning

However, there is one obvious issue in Figure 4: the middle square has two contours overlapping each other. This is likely due to the edge detector recognizing the black borders as squares (which is technically not wrong). Unfortunately, this will cloud the judgement of the location calculator and color detector, and since the two overlapping squares have very close side lengths/areas, it is not possible to prune with thresholds. To solve this, we use the contour hierarchy that was previously mentioned. Since the hierarchy provides the first child of the contour, we simply need to check if the child is in the set of contours already. If the contour does not have a child, the index would be -1.

```
# find all existing indices of contours after the first filter
contour_idx_set = set([item[2] for item in items])

# filter all contours that have children also in the filtered contours
(overlap)
# hierarchy is in format [Next, Previous, First_child, Parent]
items = list(filter(lambda item: not item[1][2] in contour_idx_set, items))
```

After this filter, we should have a clean image with all 9 relevant squares properly identified.

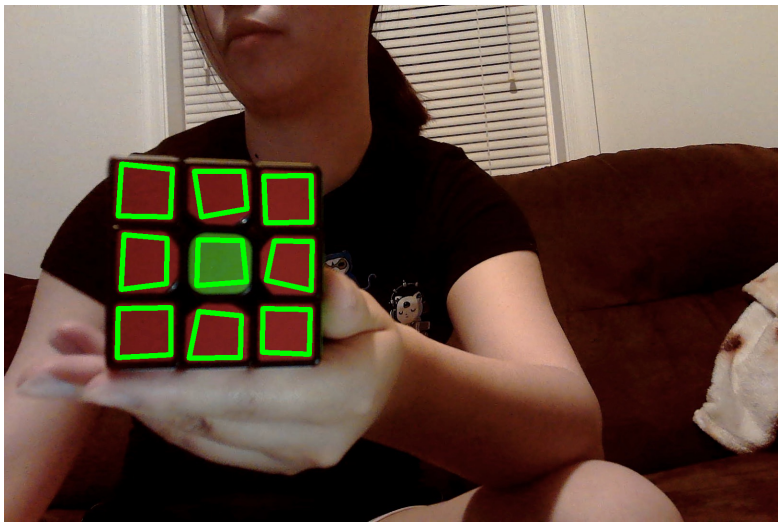


Figure 5: Contours after overlap removal

Before moving on to color recognition, we must also sort the contours so it matches the positional relations between the 9 squares. We do so by first finding the coordinates of the top left corner of the top left square, by sorting the contours by the sum of the x,y coordinates of the top left corner. Once we have this information, we sort by rows given the anchor, which is the top left square, and then sort within the rows by column.

```
def sort_contours_by_pos(contours):
```

```
    # find top left square
    contours.sort(key=lambda item: item[0][0][0] + item[0][0][1])
    first = contours[0]

    # sort by rows
    contours.sort(key=lambda item: item[0][0][1] - first[0][0][1])
    sorted_contours = []
    for i in range(0, len(contours), CUBE_SIDE):
        temp = contours[i : i + CUBE_SIDE]
        temp.sort(key=lambda item: item[0][0][0])
        sorted_contours.extend(temp)
    return sorted_contours
```

Color Recognition

Since we have identified the 9 contours corresponding to the squares, we need to identify their colors before moving on to the solving stage. The big challenge here is correctly identifying the colors under different lighting conditions. For example, yellow might look white or green under yellow light (shown in Figure 6).

Figure 6: Example of bad lighting affecting color

The general approach of color recognition is to have "pure" colors defined to compare to the input colors, and have a comparison method that outputs a score. We want to find a "pure" color that minimizes this score. Initially, we tried using the Euclidean distance as the comparison method, in other words,

$$color(x) = \text{argmin}_{y \in C} \sqrt{(x_r - y_r)^2 + (x_g - y_g)^2 + (x_b - y_b)^2}$$

```
def find_color(input_color):
    diffs = [
        np.sum(np.array([(input_color[i] - COLORS[color][i]) ** 2 for i in
range(3)]))
        for color in COLORS
    ]
    min_idx = np.argmin(np.array(diffs))
    return {i: key for i, key in enumerate(COLORS)}[min_idx]
```

However, taking the sum of the differences in each channel is too naive of a method, and it has been proven to underperform under bad lighting conditions. It has trouble differentiating between orange and red, as well as green and yellow. Colors can take many different shades under different lights: in Figure 7, 17 and 33 are supposed to be the same color. We need a more stable method to compare colors instead of just Euclidean distance.

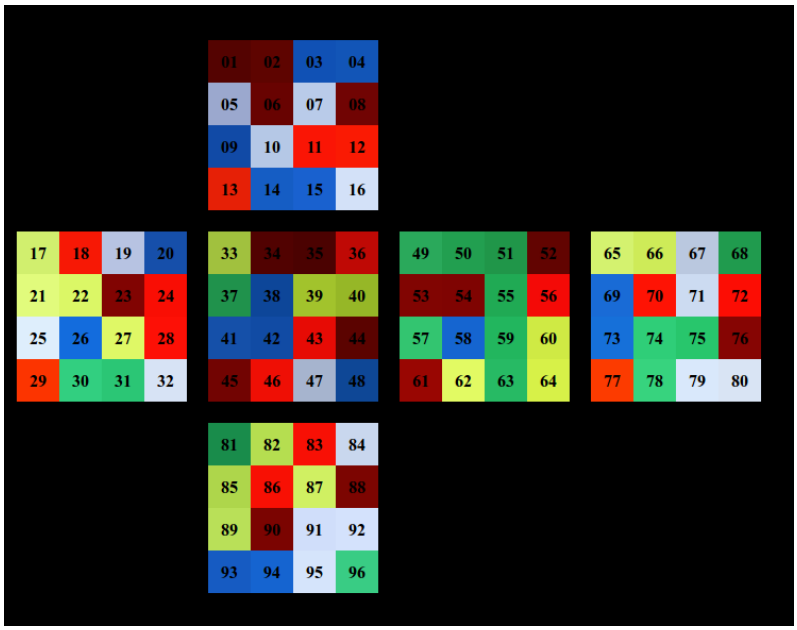


Figure 7: Same colors under different lighting conditions (Walton, 2017)

We discovered that there is a color space, CIELAB, that is designed to approximate human vision. This is more appropriate for our use case, since we do not care "what shade of red" the color is, we just want to know that it is red. The CIELAB space represents the entire gamut of human daylight vision. The three coordinates L*, A*, B* represent the lightness of the color, its position between red and green, and its position between yellow and blue. To compare colors in LAB space, we use the CIEDE2000 algorithm. The distance metric used is as follows:

$$\Delta E_{00}^{*} = \sqrt{(\frac{\Delta L'}{k_L S_L})^2 + (\frac{\Delta C'}{k_C S_C})^2 + (\frac{\Delta H'}{k_H S_H})^2 + R_T \frac{\Delta C'}{k_C S_C} \frac{\Delta H'}{k_H S_H}}$$

Where R_T is a hue rotation term, S_L is the compensation for lightness, S_C is the compensation for chroma, and S_H is the compensation for hue. Thus, we want to have

$$color(x) = \text{argmin}_{y \in \mathcal{C}} \Delta E_{00}^{*}$$

The implementation of the distance function is taken from https://github.com/lovro-i/CIEDE2000.

```
# extract color
for i in range(len(sorted_contours)):
    approx = sorted_contours[i]
    cv2.drawContours(img, [approx], -1, (0, 255, 0), 10)
    mid_x = 0
    mid_y = 0
    for point in approx:
        mid_x += point[0][0]
        mid_y += point[0][1]
    mid_x /= 4
    mid_y /= 4
    b, g, r = brightened_img[int(mid_y)][int(mid_x)]

    lab = BGR2LAB((b, g, r))

    r, g, b = COLORS[find_color(lab)]
    if i % 3 == 0:
        color_locs.append([find_color(lab)])
        center_locs.append([(int(mid_x), int(mid_y))])
    else:
        color_locs[-1].append(find_color(lab))
        center_locs[-1].append((int(mid_x), int(mid_y)))
    cv2.fillPoly(img, [approx], (b, g, r))
```

Filling in the contours with the colors we detected, we have what is shown in Figure 8.

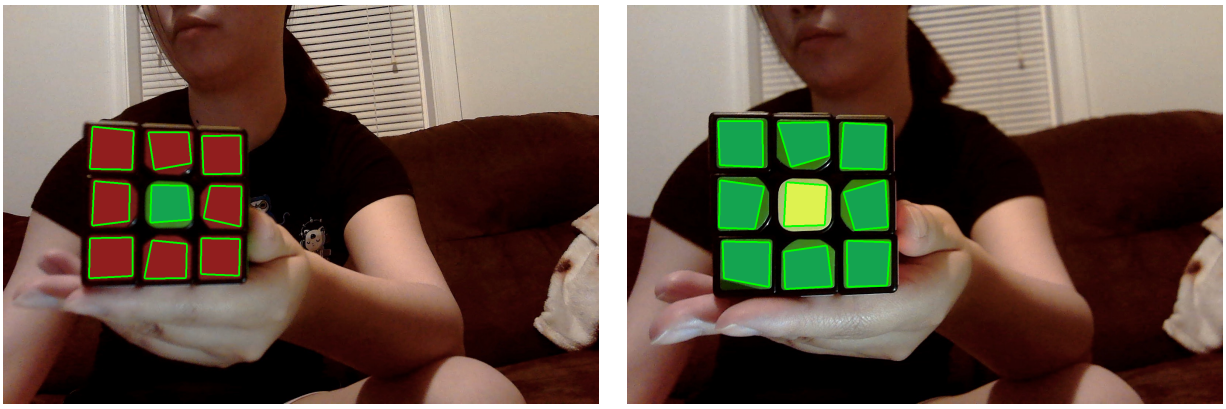


Figure 8: Contours after color detection

Video Stream Implementation

There are two main components of the video stream application. The first is the user scanning the 6 faces of the cube, and the second is solving the cube and displaying the instructions. Since the user must scan the 6 faces in the order of the expansion of the cube, as the order that is implied in Figure 7 with the correct orientations, we added an empty cube framework on the top left corner of the screen that fills up as the faces are scanned.

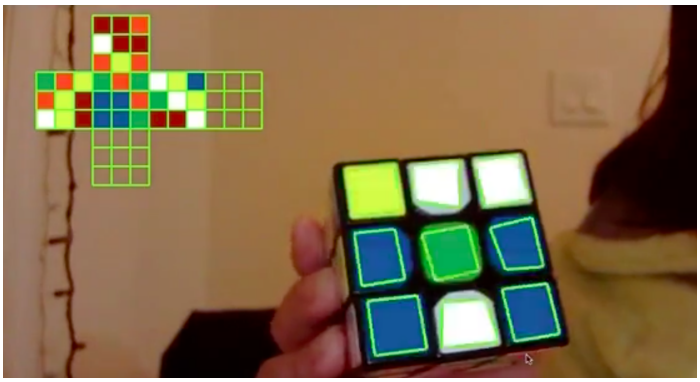


Figure 9: UI scanning phase demo

```
dilated = preprocess_img(frame1)
contours = squares_from_contours(dilated)
if len(contours) == 9:
   frame1 = increase_brightness(frame1)
   sorted_contours = sort_contours_by_pos(contours)

   color_locs, _ = color_locs_from_contours(
       sorted_contours, frame1, frame
   )
```

The UI processes the image as described in the previous parts of this section, and displays the detected colors if the number of detected contours is 9. Once the number of detected faces is 6, the app enters its solving phase, which will be discussed in the next section. The UI will then draw arrows on the screen to instruct the user.

```
dilated = preprocess_img(frame1)
contours = squares_from_contours(dilated)

if key == 3 and self.solution_idx + 1 < len(self.solution):
   self.solution_idx += 1

if key == 2 and self.solution_idx - 1 >= 0:
   self.solution_idx -= 1

if len(contours) == 9:
   frame1 = increase_brightness(frame1)
   sorted_contours = sort_contours_by_pos(contours)

   _, center_locs = color_locs_from_contours(
       sorted_contours, frame1, frame1
   )
   draw_arrows(self.solution[self.solution_idx], frame, center_locs)
```

The arrows are drawn depending on the position of the cube to further assist the user.

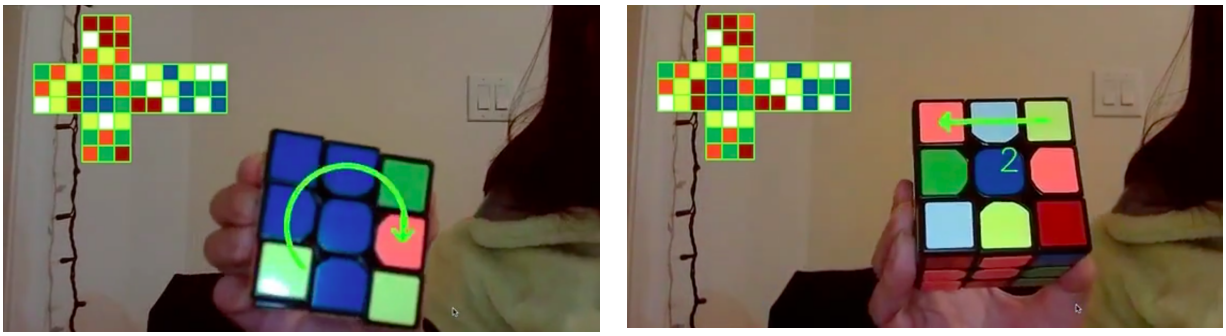


Figure 10: UI solving phase demo

Solving the Cube

Since the Rubik's cube first assembly in 1974, a number of algorithms have been found designed to solve its puzzles. The algorithms utilized by our app are computer based algorithms

that aim to find the solution with the least number of moves. As a result, these solutions are unlikely to be those found by humans trying to solve the cube under normal circumstances. During the initial experimentation phase of our project, we explored three different Rubik's cube solving algorithms, Thistlethwaite's algorithm, Korf's algorithm, and Kociemba's algorithm which we utilize in our final product. We begin this section by first explaining rudimentary Rubik's cube terminology useful in understanding the algorithms, and we will follow by exploring the three algorithms.

Cube Terminology

A standard 3 x 3 x 3 Rubik's cube consists of 6 different faces that are labeled U(p), D(own), R(ight), L(eft), F(ront), and B(ack).

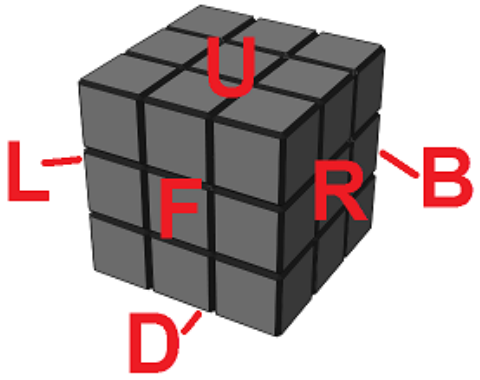
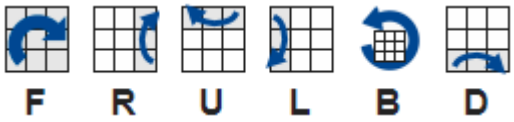
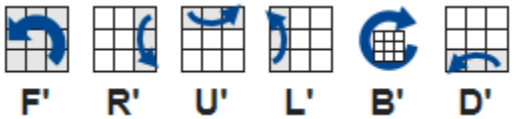


Figure 11: Notation for faces of a Rubik's cube

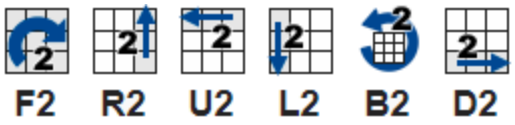
Notionally, a single letter by itself refers to a clockwise 90 degree face rotation. For example, U denotes an Up face quarter turn of 90 degrees clockwise.



A letter followed by an apostrophe refers to a counterclockwise 90 degree face rotation. For example, U' denotes an Up face quarter turn of 90 degrees counterclockwise.



A letter followed by the number 2 refers to a double turn (180 degree) face rotation. For example, U2 denotes an Up double (180 degree) turn.



Our application outputs a sequence of cube moves such as U D R' D2.

Turning the faces of a solved cube while only using certain moves will only generate a smaller subset of possible cubes. This subset can be considered a cube *group* in accordance with group theory. Most of the algorithms below utilize this fact in generating solution sequences. For example, a subset can be denoted $G_1 = \langle U,\ D,\ R2,\ L2,\ F2,\ B2 \rangle$. In this subset, the orientations of the corners and edges cannot be changed. In other words, the orientation of an edge or corner at a certain location is always the same. And the four edges in the UD-slice (between the U-face and D-face) stay isolated in that slice.

Thistlethwaite's Algorithm

Found by Morwen Thistlethwaite and published in 1981, Thistlethwaite's algorithm relies on restricting positions of the cube into groups of cube positions that can be solved with only a certain set of moves. The set of groups is given below.

- $G_0 = \langle L,\ R,\ F,\ B,\ U,\ D \rangle$

  This group contains all possible positions for a Rubik's cube.

- $G_1 = \langle L,\ R,\ F,\ B,\ U2,\ D2 \rangle$

  This group contains all positions reachable from a solved Rubik's cube state using only the moves found in this group.

- $G_2 = \langle L,\ R,\ F2,\ B2,\ U2,\ D2 \rangle$

  This group contains the positions obtainable with only the moves found in this group.

- $G_3 = \langle L2,\ R2,\ F2,\ B2,\ U2,\ D2 \rangle$

  This group contains the positions obtainable with double turns on any side.

- $G_4 = \langle 1 \rangle$

  This is the final group which consists of only the solved cube.

From here, tables of each right coset of $G_{i+1} \backslash G_i$ are pre-computed and can be used to take a cube from one group to the next. Thus, for an arbitrary cube beginning in G_0, we find the element in the right coset of $G_1 \backslash G_0$, then apply this process to the cube to reach G_1. From here,

we find the element in the right coset of $G_2 \backslash G_1$ and so on until the cube reaches G_4 where it is solved.

Kociemba's Algorithm

Kociemba's algorithm found by Herbert Kociemba in 1992 can be described as an improvement upon Thistlethwaite's algorithm. Kociemba reduced the number of intermediate cube groups to the following.

- $G_0 = \langle L,\ R,\ F,\ B,\ U,\ D \rangle$

  This group contains all possible positions for a Rubik's cube.

- $G_1 = \langle U,\ D,\ R2,\ L2,\ F2,\ B2 \rangle$

  This group contains all positions reachable from a solved Rubik's cube state using only the moves found in this group.

- $G_2 = \langle 1 \rangle$

  This is the final group which consists of only the solved cube.

As with Thistlethwaite's Algorithm, Kociemba's algorithm computes the right coset of $G_1 \backslash G_0$. However, Kociemba utilizes iterative deepening A* with a lower bound heuristic function (IDA*) to compute the shortest path to G_1. The heuristic used by the algorithm in the first step estimates for each cube state, the number of moves necessary to reach G_1. This heuristic is stored in a lookup table. It is important for the heuristic to overestimate this value since IDA* prunes branches larger than this over estimated value. From G_1, the algorithm again uses IDA* to compute the shortest path to G_2, this time constrained to the moves found in this subgroup. The heuristic used by the algorithm here only estimates the number of moves necessary to reach G_2 since the number of cube configurations in G_1 is extremely large.

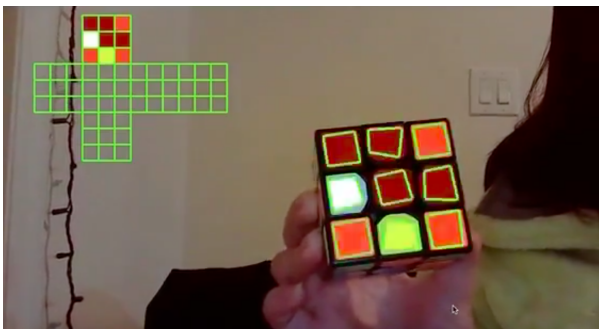
After computing one solution, the algorithm continues to search for shorter solutions by using suboptimal paths that take the cube to longer to reach G_1. Although these paths take longer to reach G_1, it is possible to make up for these by reaching G_2 quicker. The algorithm concludes if either the length of reaching G_1 from these suboptimal is too long, or if the length of reaching G_2 from G_1 is 0.

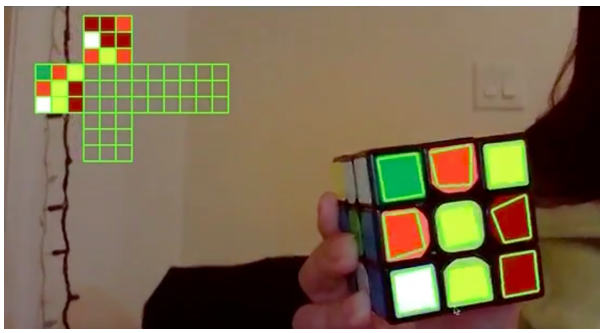
Korf's Algorithm

Korf's algorithm (R. Korf) takes a different approach to this problem. Korf's algorithm guarantees that the solution it generates requires the minimum number of moves to reach the solved cube. Korf's algorithm also utilizes IDA* to compute an optimal solution for an arbitrary cube. Korf identifies a number of subproblems small enough to be computed optimally to be utilized as a lower bound on the number of moves needed to solve the entire cube. These included, the cube restricted to only the corners without looking at the edges, the cube restricted to only 6 edges, not looking at the corners or at any other edge, and the cube restricted to the other 6 edges. Korf stores the solutions to these in a precomputed pattern database to be used as the heuristic function utilized by IDA*. Unfortunately, this algorithm has no worst case analysis, as it is unknown how many iterations this algorithm might need to converge on a solution. For this reason, our finalized app utilizes Kociemba's algorithm rather than Korf's. Kociemba's algorithm guarantees a fast runtime and typically requires less than 20 moves to reach the solved state.

Evaluation

We have run the program with various scrambled Rubik's cubes, and our program consistently provides a detailed solution within 10 to 20 steps. Here is an example of the scanning phase (the full video can be found at https://drive.google.com/file/d/1wYjh0bRD8KbtMLeGF6P8vV11IJxACVbp/view?usp=sharing):





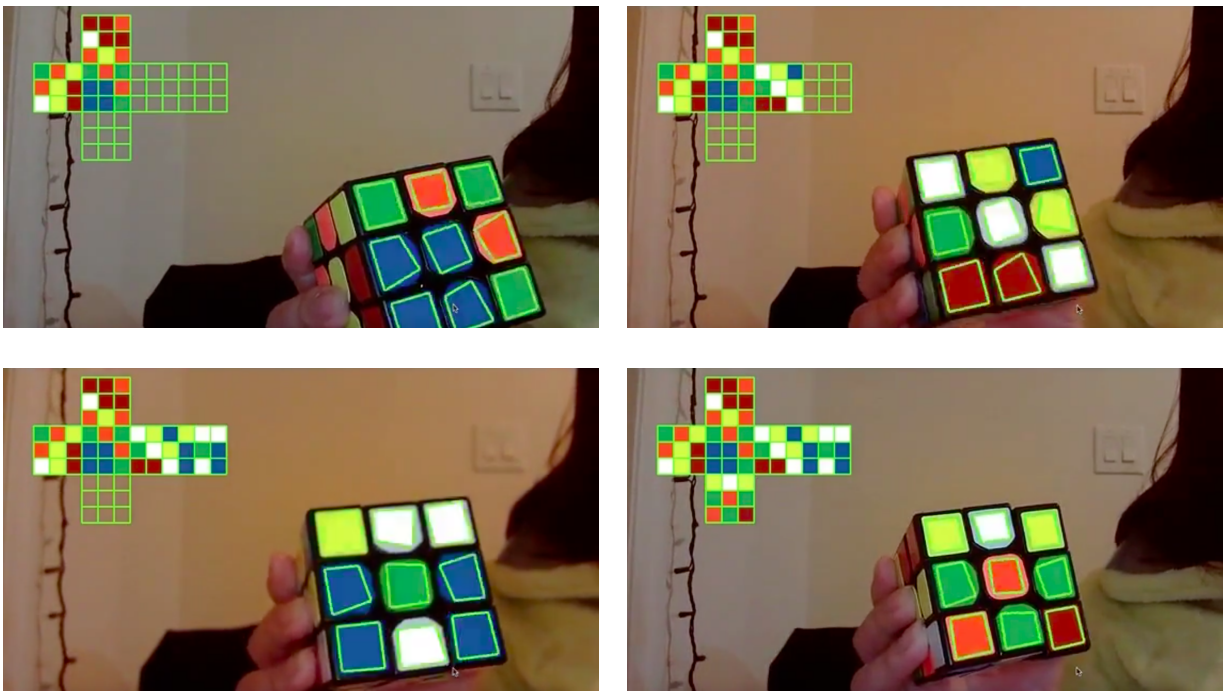


Figure 12: Example of full scanning phase

The scanning phase works very well, and exceeds our expectations, especially under bad lighting. However, it does require the user being very familiar with cube orientations, since it requires the user to scan the faces in the order of up, left, front, right, back, down. In addition, the scanning orientation has to match the structure of the cube. To make this process easier, we added the framework map to the UI, but it still requires a good amount of user understanding. In addition, this component of our project does not work well with cubes that have ambiguous or close colors. We started this project with a cube that has extremely similar reds and oranges, to the point that sometimes we cannot even differentiate the two colors ourselves under bad lighting conditions. We have tried many different methods to try to account for this situation, but could not find a solution, so we had to replace the cube with a better one. Finally, the detector is very confused when the user sits on, for example, a plaid couch, since the squares on the couch are about the same size as the rubik's cube squares. We think that this could be solved by either a deep learning algorithm to verify the detection of a Rubik's cube, or simply some sort of positional check, but we were not able to find one in this project.

Additionally, even though we are able to solve the cube in 10 to 20 steps consistently, there are times when we create a really simple puzzle by turning the cube 4 times. We can solve the cube within 4 simple turns. However, the solution produced by our application requires more than 15 steps. It is very optimal in most cases, but not the most optimal. To find the most

optimal solution, we would have to use something like Korf's algorithm. But given our use case, it is too expensive computationally and spatially. Thus, generally speaking, our application works very well, and we have succeeded in implementing all our proposed features.

Conclusion

In this project, we have developed a Rubik's Cube Solver with a computer vision-based scanning system and a solver using the Kociemba's algorithm. The scanning phase includes preprocessing the image, calculating contours, filtering contours, and color detection based on the CIEDE2000 algorithm. The solving phase utilizes Kociemba's algorithm to quickly compute a valid and short solution given the processed cube and presents to the user a sequence of moves through a visual UI. We have developed a user-friendly UI that includes a cube framework reflecting the scan status of the cube, and arrows drawn on the cube to guide the user through solving the cube. We have tested the program on various random orientations of the Rubik's cube, and are able to solve the puzzle within 10 to 20 steps every time. In terms of our proposed idea, our application exceeded our expectations. Future work for this project may include extending the app to different sizes and shapes of Rubik's cubes.

References

L. Feng, D. Jiang, A. Zhang, S. Liu, F. Wang and Y. Liu, "Color Recognition for Rubik's Cube Robot," 2019 IEEE International Conference on Smart Internet of Things (SmartIoT), 2019, pp. 269-274, doi: 10.1109/SmartIoT.2019.00048.

Bradski, G. (2000). The OpenCV Library. Dr. Dobb's Journal of Software Tools.

Walton, D. (2017). Rubik's Cube Tracker using OpenCV. LEGO Mindstorms Projects. http://programmablebrick.blogspot.com/2017/02/rubiks-cube-tracker-using-opencv.html

R. Korf, Finding optimal solutions to Rubik’s Cube using pattern databases, in: Proc. AAAI-98, Madison, WI, AAAI Press, Menlo Park, CA, 1998, pp. 700–705.