Indian Institute of Information Technology Allahabad
Department of Information Technology

Image and Video Processing Course 2021

Progress Report on

Autonomously Solving Rubiks Cubes using Image Analysis

Submitted by

IIB2019030	Kandagatla Meghana Santhoshi
IIT2019184	Pratyush Pareek
IIT2019185	R Shwethaa
IIT2019204	Mitta Lekhana Reddy
IIT2019205	Sanskar Polaki Patro
IIT2019208	Dhanush Vasa
IIT2019219	Gitika Yadav

Under the guidance of
Dr. S.R. Dubey

Abstract

In this report,we have discussed our project that aims to use image analysis to determine the state of a standard Rubik's cube using input through the live camera and then autonomously provides an optimal set of instructions to solve that Rubik's cube efficiently.

1 Introduction

Rubik's cubes have gained popularity throughout the world and most people have tried their hands at this seemingly simple, yet insanely difficult little puzzle. The Rubik's Cube is a 3-D mechanical puzzle which was invented in 1974 by Erno Rubik as a toy. A classic Rubik's Cube has 6 faces, each covered by nine stickers of one of six colors, the standard set being: green, red, blue, orange, white, and yellow. The cube has 12 degrees of freedom as each of the 6 faces can be rotated in 2 directions each. After the cube is jumbled by a series of rotations, the cuber is expected to solve it by bringing it back into its initial stage through a series of notations. In this project, we attempt to make an autonomous Rubik's cube solver that uses image processing to detect the 3x3x3 cube, recognizes its present state, and uses Kociemba algorithm to develop the optimal set of instructions required to solve the given cube.

1.1 Motivation

The 3x3x3 Rubik's cube has 43,252,003,274,489,856,000 (43 Quintillion) possible states. This number is so large that if there was one standard-sized cube representing each valid permutation, then these cubes would cover the Earth's entire surface area 275 times over. Therefore, random moves on an unsolved cube are not very likely to lead us anywhere. Motivated by the simple joy of problem-solving and on our quest to creating software-based solutions to ubiquitous problems, we decided to create an easy and intuitive software solution for everyone to solve the standard Rubik's cube.

1.2 Objective

For the scope of this project, our objective is to develop a program that detects Rubik's cube through a camera in real time, identifies its faces and provides an optimal solution. The problem can be divided into three steps: detecting the initial state of the cube using a set of images, creating a set of instructions to optimally solve the cube, and expressing the instructions to the user in an efficient and intuitive way.

1.3 Notations[1]

The most common $3{\times}3{\times}3$ Rubik's Cube notation to denote a sequence of moves is the one developed by David Singmaster, known as the "Singmaster notation". The

Singmaster Notation allows algorithms (a sequnce of standard moves) to be written in a manner that is applicable regardless of the state or orientation of the cube.

The standard notations are:

F (Front): the side faced by the solver

B (Back): the side opposite to the front side

U (Up): the side above the front side

D (Down): the side below the front side

L (Left): the side immediately to the left of the front side

R (Right): the side immediately to the right of the front side

The occurence of one of these letters in the algorithm implies a clockwise turn of the corresponding face. When a prime symbol ( ) follows a letter, it denotes an anticlockwise face turn. If the letter is followed by the number 2 (eg- F2), it denotes two turns of that face, i.e., a 180 degree turn.

1.4 Literature Review

1.4.1 *Justin Marcellienus, "The most efficient algorithm to solve a Rubik's cube"*[2]

Justin Marcellienus, "The most efficient algorithm to solve a Rubik's cube" This paper explored the history of several mainstream algorithms for solving the Rubik's cube efficiently and compared the three main algorithms: Thistlewaite, Kociemba and Korf using a series of meticulous experiments and concluded that Korf algorithm is the most efficient algorithm, closely followed by Kociemba's algorithm.

1.4.2 *Justin Ng, "Automated Rubik's Cube Recognition"* [3]

This paper explored various methods to develop a procedure that accurately identifies the state of a standard Rubik's cube. The paper explored techniques such feature detection, lighting compensation, and pixel classification.

1.4.3 *Tomas Rokicki, "Twenty-fives Moves Suffice for Rubik's Cube"*[4]

Developing on the work of Kociemba, Rokicki attempted to push the boundaries by trying to develop an algorithm to solve the Rubik's cube in 20 moves.

2 The Program

2.1 *APPLICATION*

The program is a desktop software (Windows/Linux/macOS) developed using Python 3.7 and its modules. Implementing the objective and research of this project, the program provides an intuitive User Experience and minimalist but elegant user interface to scan a 3x3x3 standard Rubik's cube using the PC's webcam and receive a well-articulated optimal solution to solve it.

The program to detect the cube and its colors relies solely on the first principles of computer vision and doesn't use a complex Convolutional Neural Network, thus demonstrating the potential of Digital Image Processing techniques even without complex ML/DL models and colossal data-sets.

2.2 ***OBSTACLES ENCOUNTERED***

The biggest obstacle against this ambition was the diversity of Rubik's cube types, the varying color palettes, the specifications of the camera and its corresponding software, and the lighting conditions of the user's environment. These variations created several problems in color detection and after trying several methods (refer to section 3.2), the team decided to add a Calibrate feature, allowing users with varying color palettes and cameras to effectively use the software.
Another major obstacle was detecting the cube's frame amidst several items of different shapes and sizes that may be in the live video frame. This was achieved just using the first principles (details in section 3.1).

2.3 ***CALIBRATION***

The calibrate mode helps the user calibrate the software and make it familiar with the colours of their cube, and their camera and lighting conditions. The user can easily access the calibrate mode by pressing 'c' and then show each center's color one by one according to the program's instructions.
The default values hard-coded into the program for each color are:

- **Red**: (0,0,255)
- **Orange**: (0,165,255)
- **Blue**: (255,0,0)
- **Green**: (0,255,0)
- **White**: (255,255,255)
- **Yellow**: (0,255,255)

As the user calibrates, the program will replace these values with the ones detected during calibration.

2.4 ***DETECTION***

The software looks for the 9-cube pattern in the live video footage (details in section 3.1.4) and when detected, highlights them by drawing bounding rectangles. The user interface includes two empty cube-face-stickers on the top-left of the screen. The top sticker shows the colors that are being detected and changes in real-time as the user moves or flips the cube. Once the user has verified that the colors of the cube have been detected correctly, they can press the spacebar to save the state of

that face. The saved state is shown on the second sticker and remains the same until the spacebar is pressed again for a different state. The bottom-right shows all the detected states on an intuitive 2D mapping of the cube.
Once all 6 sides are successfully detected, the user can press esc to close the program and receive comprehensive instructions of 20 moves or less to solve the cube easily.
To effectively scan the cube:

- We keep the white side at the top and the green side facing the camera. We start by scanning the green side.
- Once the green side is successfully scanned, we rotate the cube once (by 90 degrees) horizontally (in either direction).
- We repeat the previous step for blue, red and orange sides as well.
- Once we're back on the green side, we are in the same position where we started, with the white side at the top and the green side facing the camera. We rotate the cube forward, resulting in the white side facing the camera and the green side at the bottom.
- To scan the yellow side, we turn the cube backwards twice, resulting in the yellow side facing the camera with the green side on the top.

3 Methodology

3.1 *Cube Detection*

The first step of this project is to detect the Rubik's cube face in the live video being taken by a camera. Since there may be a plethora of different objects of varying sizes, shapes and colours in the image, it is crucial to pre-process the image and remove the extra objects before we move on to detect colors. The process to separate the cube from the rest of the objects is:

3.1.1 *Convert the image from BGR to Gray*

Since we have used CV2 in the implementation of this project, the image, by default, is stored in its BGR (Blue-Green-Red) format. We converted it to GrayScale by simply taking the arithmetic mean of the three colours for each individual pixel. GrayScale image is more suited to edge detection.

3.1.2 *Blurring the image*

The input image is very likely to have a large amount of noise among the pixels. This can make the process of edge detection extremely difficult as the noise may cause sharp changes in the grayscale values, and those will also be detected as edges. To avoid this, the grayscaled image is smoothed using a 3x3 Gaussian Blur.

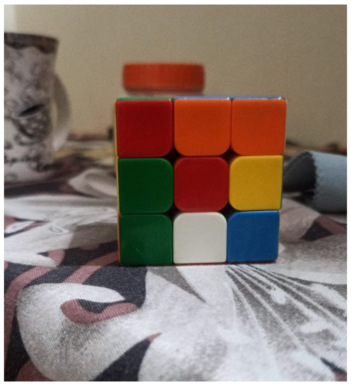

Figure 1: Input image

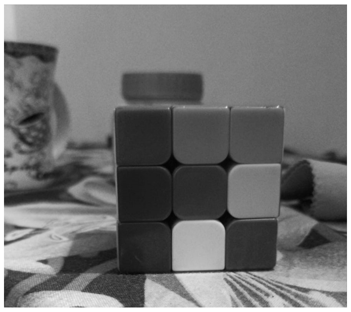

Figure 2: Grayscaled image

3.1.3 ***Edge Detection***[5]

An edge can be defined as a sharp change in the color. Detecting the edges of the cube, the individual squares on the face of the cube and the rest of the objects is the the first step in cube detection and this is done using the Canny Edge Detector.

The Canny edge detector, developed by John F. Canny in 1986, is an edge-detection operator and uses a series of operators to effectively detect a variety of edges in a given image.

1)**Gradient Calculation**

The Gradient calculation step detects the intensity of the edges and direction they're headed to using edge detection operators. To detect edges, i.e., sharp changes, we apply filters such as the Sobel Filter to detect the sudden changes in intensity. This is done for horizontal as well as vertical directions and the vector sum of the two results in the magnitude of edge at any point. The direction of the edge (the tangent at the edge) is calculated by the ratio of the gradient y and gradient x at that point.

2) **Non-Maximum Suppression**

We expect the final image to have fine and complete edges. This requires thinning out the edges and suppressing faux edges. We iterate over the gradient intensity matrix and separate the pixels with high value in their respective edge direction. The rest of the pixels are suppressed as noise.

3) **Double Thresholding**

This steps classifies each of the pixels in the gradient intensity matrix as one of the three: strong, weak or irrelevant. Strong pixels have high intensity and contribute to the final edge. Weak pixels have lower intensity than the strong pixels but not low enough to be ignored, they are dealt with in the next step. The pixels with very low intensity are classified as irrelevant and ignored.

4) **Edge tracking by Hysteresis**

Hysteresis deals with the weak pixels that were left after the last step. A weak pixel is promoted to a strong one if and only if at least one of the pixels around the one being processed is a strong one.

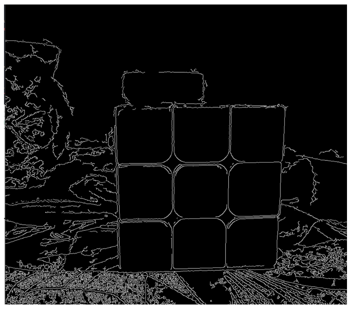

Figure 3: Canny Edge Detection

3.1.4 *Segregation*

Now that Canny Edge detector has detected all the edges, the next step is to separate the edges of the cube from the edges of the rest of the object in the picture. This is done by:

1. **Dilation** : The edges are dilated 4 times to make them more prominent and complete.

2. **FloodFill**: After dilation, the cube's face (as a whole) and each square in it would have a complete white boundary surrounding it. To get rid of most of the extra object boundaries, we can floodfill the image with white colour from each of the boundary pixel.

3. **Contour Detection**: The leftover boundaries are again dilated twice to make them more prominent and complete. Canny Edge Detection is applied again and the contours are detected and stored in an array.

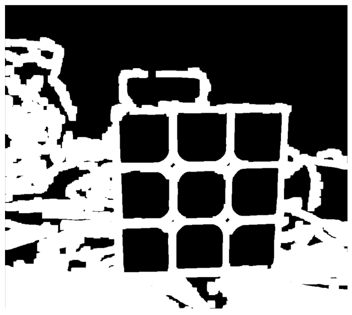

Figure 4: Dilation

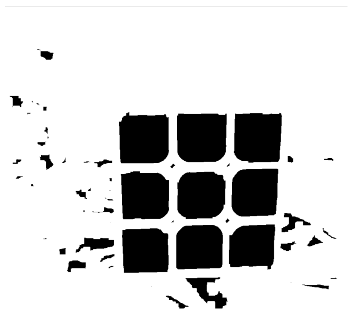

Figure 5: Floodfill

4. **Area Filter**: The Area Filter iterates over all the contours left and if their area doesn't fit within a pre-defined range (determined by the size of the input image and the expected distance of the cube from the camera), deletes them from the contours array.

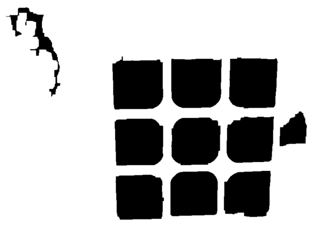

Figure 6: Area Filter

5. **Polygon Filter**: The Area Filter may still have left us with several objects that may have the same area that we expect from the squares in our cube's face. We can list all the contours and those which do not approximate as a square (with a margin of error) are removed from the contours' array.

6. **Neighbourhood Filter**: All these filters may still leave behind several small squares of the size expected from a Rubik's cube's square. To segregate the

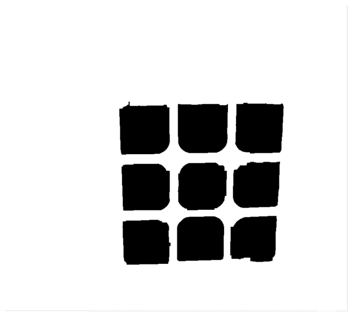

Figure 7: Polygon Filter

extra squares, we can iterate over each contour left in the array (if it's less than 9, then detection is unsuccessful and starts over) to check which of them has 8 neighbours.
To do this, we create 9 positions for the current contour which are the neighbors. We'll use this to check how many neighbors each contour has. The only way all of these can match is if the current contour is the center of the cube. If we found the center, we also know all the neighbors, thus knowing all the contours and thus knowing this shape can be considered a 3x3x3 cube. When we've found those contours, we sort them and return them.
Now that we know how many neighbors all contours have, we'll loop over them and find the contour that has 9 neighbors, which includes itself. This is the center piece of the cube. If we come across it, then the 'neighbors' are actually all the contours we're looking for.

7. **Reducted image**: Finally, the image is reduced to just the bounding rectangles detected in step 5 and now only consists of the 9 squares of the face of the cube.

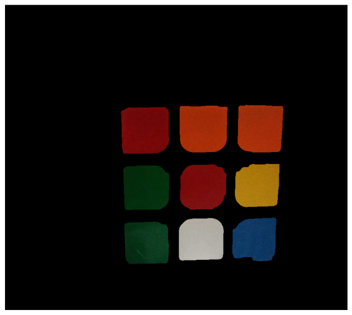

Figure 8: Reducted Image

3.2 *Color Detection*

Now that we have the dimensions of the 9 squares, the next step is to detect the colours. This is the trickiest part of the project since a cube may have a different set of shades. Even if they are the same colour for us humans (White, Red, Blue, Orange, Green, Yellow), hard coded color ranges may fail to identify them. The colours are also subject to the lighting conditions of the room.

- **Prominent Color Palette(Before Calibration)**:
  - **Red**: (0,0,255)
  - **Orange**: (0,165,255)
  - **Blue**: (255,0,0)
  - **Green**: (0,255,0)
  - **White**: (255,255,255)
  - **Yellow**: (0,255,255)
- **Identification of Dominant Color**:
  Here, we will use the help of the k-mean clustering to identify the dominant color in a certain region of interest. Hence what happens is that for the region of interest we reduce the number of colours to 1 while preserving the overall appearance quality of the region. This is achieved by first defining the termination criteria for the k-mean clustering followed by performing the K-mean clustering with the help of cv2 and retrieving the labels and palette. Next we get a array with unique labels with there respective counts stored to counts. Lastly we create a tuple of the palette values with the label with max(counts) as palette[max(counts)] and return this.
- **Identification of Closest Color**:
  Identification of the closest colour of BGR color using 2 key functions.
  - **bgr2lab[6]**:
    What this function does is that it converts a BGR colour to LAB form. Hence what happens here is that LAB is a conversion of the same information to a lightness component L*, color a* and color b*. This is done so that we can keep the lightness kept apart from color, and that can be adjust one without affecting the other along with being the best at preserving the colors.
    From :
  - **ciede2000[7]**:
    This function calculates how similar two colors are, this function was developed based on the paper "The CIEDE2000 Color-Difference Formula: Implementation Notes, Supplementary Test Data and Mathematical Observations", by Gaurav Sharma, Wencheng Wu and Edul N Dalal.

  Now that we know both this functions. We convert the BGR received into the getclosestcolor where we convert the BGR to LAB with the bgr2lab function and store to LAB. Next we compare the LAB color to the cube color palette and calculate the distance of the two colors with LAB and each colour in palette using ciede2000 and we find the colour with the least distance. Where we return that colour as the closest color.

- **Failed Methodologies**:
  Before narrowing down on the aforementioned complex methodology, we tried several methods that were simpler and more intuitive, and built upon them or changed them to improve the color detection in the program.

  - **BGR Ranges**:
    We tried detecting colours using a plethora of BGR ranges. Each square had its own color representative (started with average color, after whose failure moved on to dominant color) which was compared to different ranges and was assigned to one of the expected colors accordingly.
    These colour ranges were picked from several reliable sources but still failed to cover the nuances such as the lighting conditions, variety of cubes and the limitations of the BGR format.

  - **BGR Closest**:
    We tried detecting colors by finding their 'distance' from the standardized colors. Just like the previous attempt, each square had its own color representative (started with average color, after whose failure moved on to dominant color) which was assigned a color according to its variation from the 6 standard colors.
    This improved the distinction between some colours but failed in classifying as Orange-or-Red and Green-or-Yellow, i.e., the colours with close or overlapping BGR values.

  - **HSV Ranges**:
    BGR values detected by cv2 were converted to the HSV format and the dominant and average color were checked against pre-defined HSV ranges for each of the standard 6 colors. These colour ranges were picked from several reliable sources but still failed to cover the nuances such as the lighting conditions, and the variety of cubes.

  However, the addition of K-Means Clustering coupled with Lab format and the Calibration mode almost perfected the color detection.

3.3 *Determining the state of the cube*

In this step, we determine the state of the cube using the states of the 6 faces.
The contours that have been detected for each face are sorted according to the position of their centers and their colours are stored in 6 matrices of 3x3 each. The program also checks various restrictions related to the positions of colors. These restrictions include opposition (red side must be opposite to orange side etc), uniqueness (no repeated pieces), and count of colours (the user scanned 9 tiles for each side, that is 54 tiles for all 6 sides, etc.)

It is arbitrarily standardized that while solving, the user shall face the Green-centered side with the White-centered side on the top. This allows the normalization of the centers to the symbols: U(Up), R(Right), F(Down), L(Left), B(Back). This normalized form is used for the generation of the solution. The substitution is URFDLB (white, red, green, yellow, orange, blue).
This makes the project open to extension to various color palettes other than RGBYOW and to other type of 3x3x3 cubes and welcoming to open-source developers.

3.4 *Solving the cube*

The most important part of this project, this step will use Kociemba's algorithm that uses iterative deepening to find the optimal solution to the cube. Herbert Kociemba, a mathematics teacher and professional cuber, reduced the number of groups to only two therefore making a substantial decrease in required moves to a maximum of 20 moves.

The term "God's number" is sometimes given to the graph diameter of Rubik's graph, which is the minimum number of turns required to solve a Rubik's cube from an arbitrary starting position (i.e., in the worst case). Rokicki et al. (2010)[8] showed that this number equals 20. The bounds on God's number are known. We can find the lower bound[9]: The first twist has 12 possibilities (clockwise and counter-clockwise for each face), and the next moves can twist another face in 11 ways (while excluding the one that reverses our previous move). Using this, we can calculate the lower bound for the maximum number of moves needed to get from starting state to any state. For this, we use the "Pigeonhole Principle", i.e, the number of possible outcomes of rearranging must be greater than or equal to the number of permutations of the cube: 12 x $11^{(n-1)} \geq$ 4.3252 x $10^{(19)}$

which is solved by n $\geq$ 19.
Similarly the upper bound is found to be 20.

Cube string notation[10]
The names for the faces of the cube (the following letters stand for Up, Down, Left, Right, Front, Back):

A cube definition string "LBU..." means that in position U1 we have the L-color, in position U2 we have the B-color, in position U3 we have the U color etc.

So, for example, a definition of a solved cube would be
UUUUUUUUURRRRRRRRRFFFFFFFFFDDDDDDDDDLLLLLLLLLBBBBBBBBB

4 Datasets

We used an open-source database from Kaggle by the name of "Rubik's Cubes Faces" It is a collection of random high-quality pictures of various 3x3 scrambled Rubik's speedcubes.

5 Technology Used

This project has been implemented using the Python 3 programming language. Libraries used are:

1. **OpenCV** : OpenCV-Python is a Python library consisting of computer vision tools. We have used CV2 in this project.

2. **Numpy** : NumPy is a Python library that adds support for multi-dimensional matrices and the mathematical functions associated with them.

3. **iMutils** : iMutils is a Python library consisting of functions that help in translation, rotation, resizing, sorting contours of images.

4. **i18n**: i18n is a simple translation module with dynamic JSON storage.

5. **Kociemba** : This module contains the python implementation of Herbert Kociemba's two-phase algorithm for solving Rubik's Cube.

6 Results and discussion

- The lighting of the room plays a major role in the detected colours and even edges.

- It was much easier to detect cube with thick internal boundaries and timid colours as compared to those without internal boundaries and deeper colours.

- The most difficult step was to remove other objects from the image, specifically if they were also squares. This was eventually achieved by detecting neighbouring contours and checking if they matched the format of a cube.

- Lab format performed significantly better than HSV format which performed better than BGR when it came to detecting the color of a square and classifying it.

- It proved significantly difficult to detect cubes whose color stickers had been scratched or had logos on them because they created internal contours that messed up the cube structure when dilated. Thus, such cubes weren't used in testing.

- Color detection improved significantly after adding a Calibration mode to orient the software to the lighting conditions and camera.

- Color classification improved significantly by using K-means clustering.

7 Conclusion

- This project is helpful for solving a 3x3x3 rubik's cube. When a unsolved rubik's cube is displayed in front of the webcam or camera , it scans all the scans faces one by one.
- After successful scan i.e; when colors are detected properly and the program will start to show all the guidelines step by step to solve the cube.We used kociemba algorithm to be able to solve the cube within 20 steps or instructions.
- The calibration mode adapts the software to the lighting and camera conditions of the user and the variety of the cube.
- Colors are converted to the Lab format and K-means clustering is done on each square's dominant color to classify it as one of the 6 colors of the cube.
- The software has been designed such that it can be extended to several other 3x3x3 cubes, including but not limited to the mirror cube, diagonal cube, and many more.
- CNNs and large datasets aren't always necessary to solve image-processing problems. Softwares can be hard-programmed more specifically to the task and may outperform soft-programmed models.

8 Contribution

- **Name: Kandagatla Meghana Santhoshi**
  Designed and developed the Calibrate Mode
- **Name: Pratyush Pareek**
  Implemented and experimented with methods for cube detection and segregation
- **Name: R Shwethaa**
  Project design, compilation and configurations
- **Name: Mitta Lekhana Reddy**
  Implemented Rubik's Cube's Data Structure and Cube Verification
- **Name: Sanskar Patro Polaki**
  Developed the UI and UX to scan the cube in an intuitive way
- **Name: Dhanush Vasa**
  Implemented and experimented with several methods for color detection and classification
- **Name: Gitika Yadav**
  Researched Kociemba algorithmn and developed a system of instructions to solve the cube

References

[1] https://www.math.ru.nl/OpenGraphProblems/Kris/page1.html

[2] Justin Marcellienus, *"The most efficient algorithm to solve a Rubik's cube"*

[3] Justin Ng *"Automated Rubik's Cube Recognition"*

[4] Tomas Rokicki *"Twenty-fives Moves Suffice for Rubik's Cube"*

[5] https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

[6] https://stackoverflow.com/a/16020102

[7] https://github.com/lovro-i/CIEDE2000.

[8] Tomas Rokicki et al *"The Diameter of a Rubik's Cube group is 20"*

[9] MIT *"The Mathematics of a Rubik's Cube"*

[10] https://nerya21.github.io/MindCuber/javadoc/twophase/Facelet.html

[11] http://kociemba.org/twophase.html