University of Toronto Mississauga
CSC420 Winter 2019

Project Report: Sriram Srirangam 1001823930, Kamran Shahid 1002352189

Introduction:

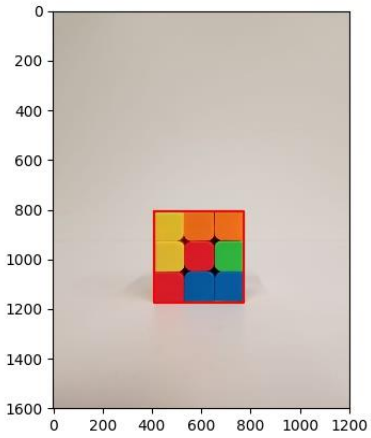
The task of solving a Rubik's cube is one that has been of interest to mathematicians and puzzle enthusiasts alike for many decades. While a plethora of Rubik's cube solvers do exist currently, they require manually entering in the configuration of the cube, one sticker at a time, which means the user has the arduous task of inputting data 54 times for a 3x3 cube (e.g. 54 clicks, a 54 character string, etc.). Given this data input, the algorithms make use of sophisticated mathematics (namely Group Theory and Combinatorics) to figure out the sequence of moves required to bring the cube to a solved state; this is not a computer vision task (see this link if you're interested in how it works).

Our objective for the project is to automate the first part of the Rubik's cube solving pipeline using concepts we covered in CSC420. Given a photo of each face of a 3x3 Rubik's cube, the goal of our application is to output the exact configuration of the cube and instructions for how to solve it, regardless of the color layout of the cube.

This project required critical thinking and creativity in deciding which in-class concepts can be applied and how, and this was different than other standard projects, which explicitly defined which steps are to be used. This meant our group had to put in effort in the trial and error stage of attempting different techniques to optimize our algorithm.

High-level Approach:

1) Figure out the 4 outer corner locations from the cube in the original image
    - There were a number of ways to go about this
    - Harris Corner Detection, SIFT, Canny's Edge, Hough Transform and searching for connected right angles were all considered
    - Canny's Edge Detection was chosen (see challenges section for why we felt the other methods were not appropriate)
    - After running our implementation of Canny's Edge Detection, we searched through all of the edge pixels and found the corners by taking the x and y minimum and maximum points to form the corners
    - This method worked well when the only object in the photo was the Rubik's cube – having other objects in the photo created outer edges outside of the Rubik's cube area

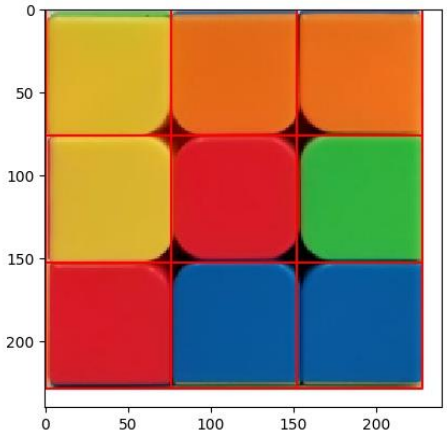


2) Taking the corners detected from Step 1, we use a Homography Transformation to get the object in a front facing perspective

- This is done in order to have the object in an enclosed bounded box where we can calculate the mean RGB value for each sticker on a face in a robust manner
- A standard Rubik's cube face has dimensions 5.7cm x 5.7cm, and these were the dimensions we selected for transforming the initial image bounded by the 4 corners

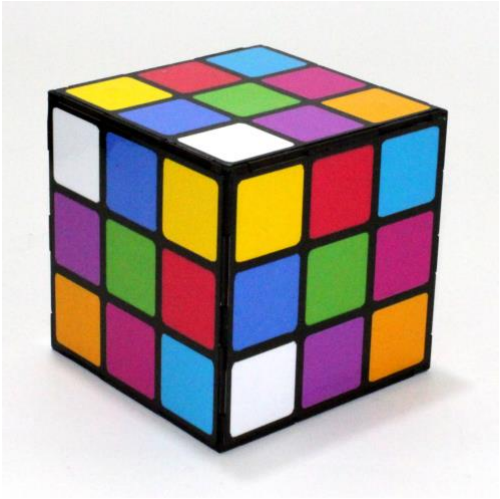
3) Calculate the mean RGB value in each square

- With our front-facing image from Step 2, we can divide the image into 9 squares of equal sizes. Based on the dimensions being 5.7cm x 5.7cm, an individual square would only be a third of the height and width -- so 1.9cm x 1.9cm.
- We iterate over each of these 9 squares, and at each square, we iterate over each pixel in the square's width and height to get a RGB tuple at each pixel location. We store each of these tuples in a list (block_colors), and then compute the pointwise average of the list of tuples (i.e. average all the R values, average all the G values, and average all the B values). This average RGB value is our 'color' for the square.



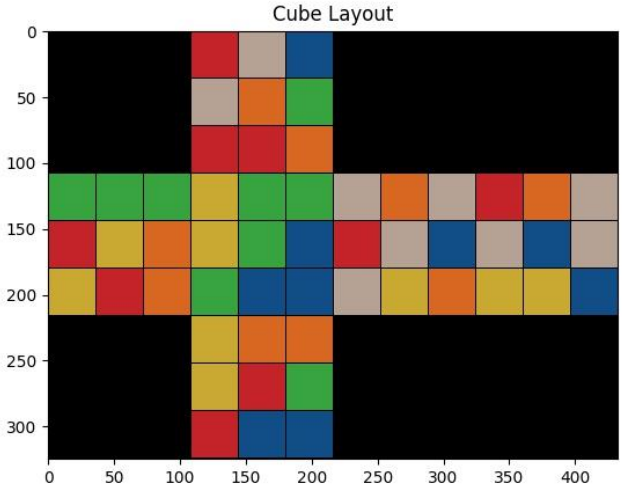
4) k-Means Clustering

- Perform k-Means clustering on the list of 54 RGB values, with k = 6, to find centroids of the dataset, which are in the form of RGB tuples. Provide each centroid with one of 6 labels.
- Calculate each Euclidean distance in R^3 space between each of the 54 colors and each of the 6 cluster centroids. Take the minimum of these distances, and the cluster centroid with the minimum distance is the RGB value which that square will take on.
- We chose to use k-Means as opposed to a more straightforward color detection technique (e.g. thresholding RGB channels, HSV transformation, etc.) because it is invariant to the actual color layout of the cube. What this means is that we don't restrict the colors of the cube to being the traditional red, orange, yellow, green, blue, white. As long as there are 6 distinct colors, our algorithm would work even if the cube has stickers that are purple, brown, black, etc.



5) Plot the cube layout and generate the input for the solving algorithm

- With the 6 centroids we found in the previous step, we label each of the squares accordingly using the Euclidean distance metric outlined above.
- Now we can plot the layout of the cube, where the color for each square in the plot is the color of the centroid which represents the cluster that square belongs to



- The solving algorithm takes in a 54 character string, where each of the characters represents one of the six colors of the traditional Rubik's cube - 'y' for yellow, 'b' for blue, 'r' for red, 'g' for green, 'o' for orange, and 'w' for white. The indexing of the string corresponds to the indexing of the following visual representation of the cube:

```
                 ---------------
                 | 0  | 1  | 2  |
                 ---------------
                 | 3  | 4  | 5  |
                 ---------------
                 | 6  | 7  | 8  |
                 ---------------
-------------------------------------------------------
| 9  | 10 | 11 | 18 | 19 | 20 | 27 | 28 | 29 | 36 | 37 | 38 |
-------------------------------------------------------
| 12 | 13 | 14 | 21 | 22 | 23 | 30 | 31 | 32 | 39 | 40 | 41 |
-------------------------------------------------------
| 15 | 16 | 17 | 24 | 25 | 26 | 33 | 34 | 35 | 42 | 43 | 44 |
-------------------------------------------------------
                 ---------------
                 | 45 | 46 | 47 |
                 ---------------
                 | 48 | 49 | 50 |
                 ---------------
                 | 51 | 52 | 53 |
                 ---------------
```

  Additionally, the string must adhere to the constraint that the center indices of each face always have the following assignment: 4 -> yellow, 13 -> blue, 22 -> red, 31 -> green, 40 -> orange, 49 -> white.
- We then assign the appropriate character to each of the other indices in the string such that all indices with the same label have the same character assigned to them. Note that the colors of the squares in the plot from above don't necessarily match the string character colors. This is because the string character colors are independent of the actual RGB colors of the squares.
- Once we have our string, we can call the solving algorithm and format the output so that the user can go over the solution step-by-step.

Work Breakdown:

Most of the collaboration on the project happened in person, where both members were actively participating in outlining/implementing steps together, so both members have a full understanding of each step of the algorithm. That being said, the division of labor is roughly as follows:

- Kamran implemented the Canny edge detection to find the four corners of the cube
- He also used these corner points to set up the homography and transform the image
- Once the image was transformed, he found the mean RGB value for each square
- Sriram then implemented the color detection for the algorithm using k-Means and used the centroids to label the mean RGB values
- He then used these labels to generate the layout of the cube and format the input for the solving algorithm

Challenges Faced and Limitations:

1) Corner Detection

As discussed above, there were several options available for detecting the outer corners of the Rubik's cube in Step 1.

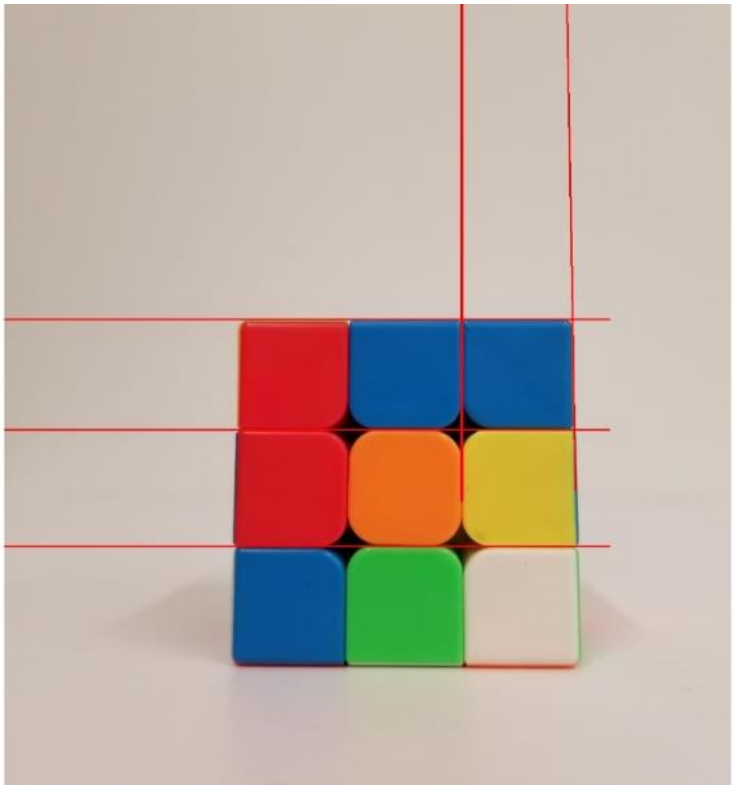
Our first approach was to use Harris's corner detection because this was a technique specifically designed for corner detection. However, in the Rubik's cube, corners were detected in between sticker areas, and were not picking up the outermost corner areas, despite adjusting threshold values. This is shown in the image below.

Another option we explored was using SIFT interest points to find the outermost corners of the cube and using feature matching to match the cube to a template. This option failed

because a Rubik's cube has similar features throughout, and the same feature would often be detected several times, leading to several incorrect matches. Furthermore, the points we were looking for (outer-most corners of the cube) were not being robustly detected from image to image, and this meant that extracting interest points for corners using SIFT was not a robust solution. See the images below for an illustration of this.

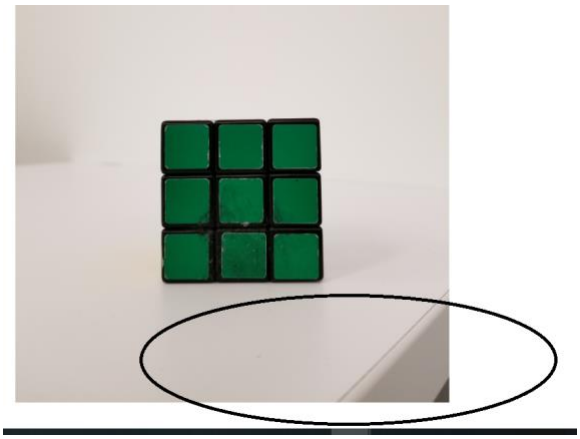
Hough Transform was another option our group seriously considered for Step 1 because it is designed for object detection within classes of shapes like squares, so it made sense because a cube has very strong, defined edges. Our initial Canny Edge solution worked well when the input images had no background noise, but failed when there were objects with edges in the photo, such as a table edge. We intended on using Hough Transform to find strong lines passing through a strong interest point, and then keep lines which form a

square by checking to see if the lines meet at 90 degree angles (corners) by examining the gradient direction, something which we did in our Assignment 1. If lines intersect at 4 corners, then this is our square we can apply homography to. In theory, we felt this idea worked, but after applying the OpenCV functions for Hough Transformation, we weren't reliably obtaining lines for all four outer edges of the cube despite adjusting parameters. This is why we decided not to implement Hough Transform ourselves, because it may not have worked. Also, we only covered Hough Transforms at the very end of the course (the final lecture to be exact), and at that point, our group was well into the project, so it would have taken a lot of refactoring to accommodate the change.



2) Requiring a background with no noise

As discussed above, by using Canny's Edge Detection, input images are required to have only the Rubik's Cube without any other significant edges in the background. Having strong edges outside of the Rubik's cube area will show up in our Canny result, and our algorithm may mistakenly detect these as corner points. For example, the below picture of a Rubik's cube has a partial view of a table edge, and despite changing Canny parameters, this table edge caused our program to detect it as a potential corner point.



While our proposal, which Professor Fidler approved, indicated that our project will require images without significant corners in the background, this is still a limitation that we feel can be improved in future iterations of our application.

3) Requiring the photos to be in the order prescribed by the solving algorithm.

   In order for the algorithm to be able to delineate which faces are adjacent/opposite to each other, we require that the photos must be either passed in in the order prescribed (i.e. top, left, front, right, back, bottom) or they be labelled as such. While this is a requirement which can't really be circumvented, it's still worth mentioning when discussing rigidities that the user may experience.

Conclusion & Next Steps:

In conclusion, we felt our project was a success. By interactively connecting course concepts in a creative process to solve a real world problem, and defining our own solution, we felt this project helped us improve our knowledge of course concepts. Given input images, our algorithm produces solution steps in a robust manner.

In the future, there is room for improvement in the strict requirements for the input images of the Rubik's cube faces passed in. At the moment, input images are restricted to having little to no background noise. In the future, we plan on exploring Hough Transform to see if this technique can detect cube corners from images with significant background noise. Furthermore, our solution at the moment only accepts 3x3 cubes at the moment, but can easily be extended to solve 4x4 or cubes that are even larger, so this might also be a promising extension for our project. Lastly, working with videos frame-by-frame rather than requiring image files can enhance user experience significantly, and is worth considering for the future.

References:

- The code for the solving algorithm can be found here: https://github.com/Wiston999/python-rubik

- We found the dimensions for estimating the size of the cube here: https://en.wikipedia.org/wiki/Rubik%27s_Cube#Mechanics

- The image for illustrating different colored cubes (item (4) in section High-level Approach) was found here: https://www.martinsmagic.com/allmagic/stage/color-changing-rubik-by-tora-magic-company/