REAL-TIME RUBIK'S CUBE DETECTION USING K-MEANS

Alan Bruner, Jiafan Lin, Luke McNeil, Leding Ren

CSSE463 Image Recognition

18 February 2022

ABSTRACT

The ability to accurately identify features of an image within a given pattern is one with a variety of usages. To demonstrate these concepts, we created a system that can identify the colored squares on a Rubik's Cube from a series of images and then generate a solution. The system created uses simple color thresholding to classify colors into one of six groups. With this system, we were able to achieve an accuracy of 92.6% sticker classification. To improve accuracy, we calibrate the system to the specific lighting conditions of the image. This improved the classification accuracy to 99.1%. However, there were still limitations on the orientation of the cube in the image.

1. INTRODUCTION

The Rubik's Cube is one of the most well-known puzzle toys, notorious for being a simple design that is difficult to solve for an average person. As such, the ability to solve a Rubik's Cube is a benchmark for many systems' ability to solve problems. For most people, the easiest way to learn to solve a Rubik's Cube is by finding online guides that show algorithms, or a series of moves, to manipulate the cube into specific states. A much easier way would be to feed images into a system and have the system output the exact steps needed to get to a solved state.

While a system to generate a solution from images of a Rubik's cube is a problem that already has numerous solutions, many of these solutions lack versatility, relying on a standard lighting setup and camera angle to work. This can lead to issues when the cube is photographed under differing lighting conditions, shown in **Figure 1**. In this paper, we developed a system that can recognize the faces of a Rubik's cube in nearly arbitrary lighting setups by calibrating the system to the specific lighting context that the images are taken under. This allows the system to be much more versatile than many other attempts at image-based Rubik's Cube solvers.

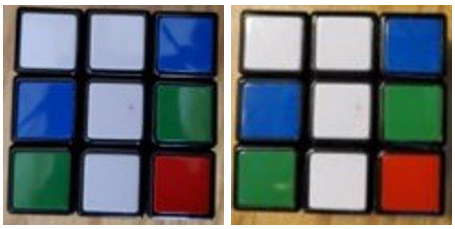

Figure 1: Rubik's Cube photographed under differing lighting conditions

The problem of generating a Rubik's Cube solution may at first seem like little more than a novelty, but the approaches taken to solve such a problem can be easily extended into much more practical use cases. For instance, taking pictures of items on an assembly line could require identifying specific color patterns on the objects that will be in specific orientations. Using a color-recognition system with calibration similar to the one used in this system would expedite the process of analyzing these parts for defects.

2. LITERATURE REVIEW

The first paper [1] we found uses CNN to find the center and size of the cube. The CNN can be trained to create a bounding box around a cube and remove unnecessary background. It also mentions using KMeans to extract superpixels and SVM to predict each superpixels identified by KMeans. In our project, we also used KMeans to extract and average colors in squares on one side of the cube. Since we did not have a training process, we did not use a CNN or SVM to train the recognizer. The second paper [2] we found mainly discussed using YUV space to normalize the pixel intensity in images. We did not utilize the YUV space in our project, because RGB space turned out to work well. Both the third [3] and fourth [4] papers mention using different mathematical approaches like Delaunay Triangula

-tion and weak label hierarchic propagation in HSV space to recognize different sides of a cube in an image. They might be useful if we want to improve this project in the future by only inputting two images with all six sides of a cube and recognizing a whole cube. In fact, we were inspired by ideas in Fruits Finder and decided to extract a mask of all 9 squares and their colors with KMeans.

3. PROCESS

3.1 Data Source

This project was different from many other image recognition tasks in that it did not use a training set. Instead, we developed the system by live testing with our webcams to determine what worked and what did not. As part of our results though, we do create a test set to see how our system performs.

3.2 Preprocessing

Before trying to identify the colors of the stickers in an image, the image is first downsampled. This makes the system much faster and realizes real-time applications with the webcam. We always resize the image to have a width of 128 pixels. With the help of image downscaling, the processing time of each frame decreases from 6 seconds to 0.1 seconds. Even though this process loses some detail from the image, it is still possible to get accurate predictions as the colors on the cube's stickers are reasonably clear.

3.3 K-Means Clustering

The first thing we did with the resized image was run the K-Means clustering algorithm. This algorithm tries to separate each pixel in the image into one of K groups by the RGB values of that pixel. This process works by iteratively improving these categories. K-Means result is an array with the exact dimensions as the image where each entry is a number from 1 to K, telling which cluster that pixel is a part of. This result can then be used to set the color of all pixels in a group to the mean color of that group. The results of this process can be seen below for a couple of values of K.

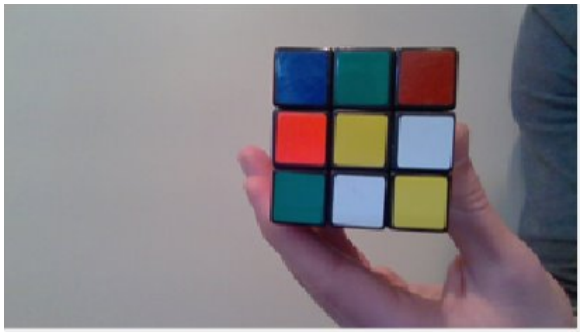

Figure 2: Original Image before K-Means

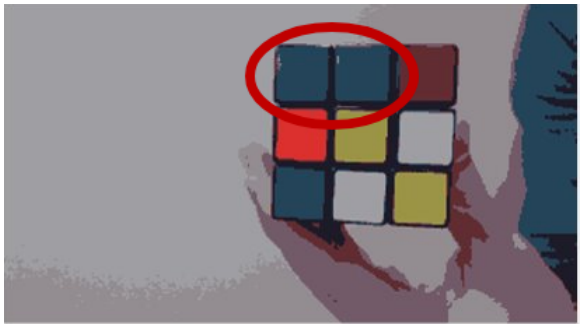

Figure 3: K=8 for K-Means

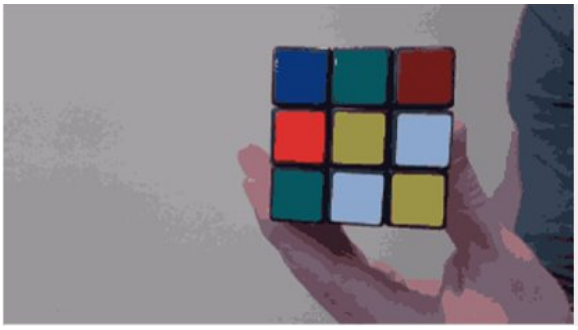

Figure 4: K=17 for K-Means

These images explain why we use K=17 for our Rubik's cube recognition. If a smaller value is chosen, then the K-Means algorithm often combines similar colors such as blue and green. This effect can be visualized in **Figure 3**. For this reason, a higher value such as K=17 works well. Doing K-Means with this K value makes the image simpler and reduces the complexity to work with for later steps. This is illustrated by the solid colors of all stickers seen in **Figure 4**. In **Figure 2**, however, each sticker comprises many slightly different RGB values.

3.4 Finding Sticker Mask

The next step is to use this K-Means image to create a mask that shows nine connected components - one for each sticker. We first consider all groups of pixels with the same values from the K-Means image to be stickers. We filter this group by only taking groups within a certain area threshold. Based on our webcam resolution and image downscaling, we accepted an area between 70 and 250 pixels based on our webcam resolution and image downscaling. The searching list is further narrowed down by getting rid of regions with a squareness of less than 0.8. Squareness is defined to be the percentage of the bounding box that the shape fills. Perfectly oriented rectangles will have a squareness of 1, while other regions have a squareness of less than 1. This restriction does a good job of throwing out many regions that looked the same size as stickers but were actually not. Below is an example of finding this sticker mask.

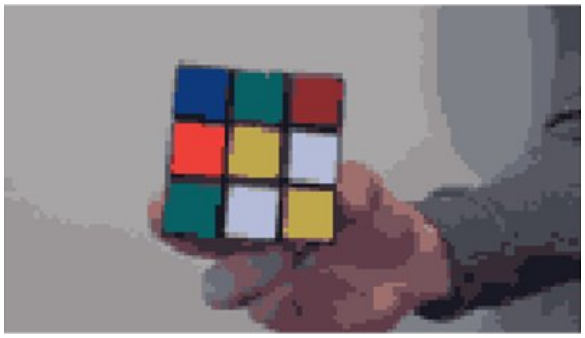

Figure 5: K-Means Image

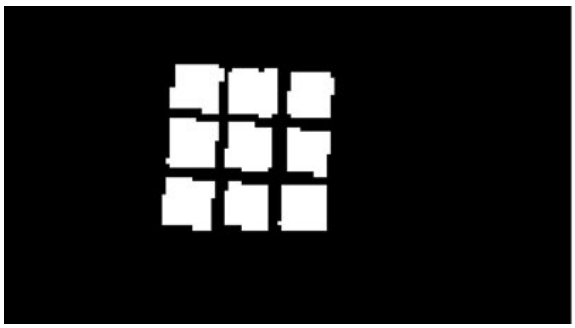

Figure 6: Sticker Mask

Figure 7: Original image with any pixels not in the Sticker Mask set to black

3.5 Getting Face of Cube from Sticker Masks

The next step is to take the sticker mask and produce a 3x3 matrix representing the color of that face of the Rubik's cube. This process assumes that exactly nine regions are found in the sticker mask. If there is any number of regions other than nine, the algorithm simply does not produce a prediction.

In the case that there are nine regions, we look at each region. To find which color a given sticker is, we compute the Euclidean distance between the color of that region from the K-Means image and six hardcoded values representing the colors of the six common Rubik's cubes colors - red, green, blue, orange, yellow, and white. Whichever color is the closest to the sticker value, we assign the block as the color of that sticker.

	R	G	B
Red	123	38	16
Green	12	120	84
Blue	25	50	190
Orange	213	106	40
White	160	180	220
Yellow	165	150	10

Table 1: Starting color values

After repeating for all nine stickers, we get a 3x3 character array that looks like the following.

Y	W	G
W	Y	O
R	G	B

Table 2: 3x3 Array representing a face

We then use a Matlab library [5] to show a 3D visualization of this prediction of the state of the cube.

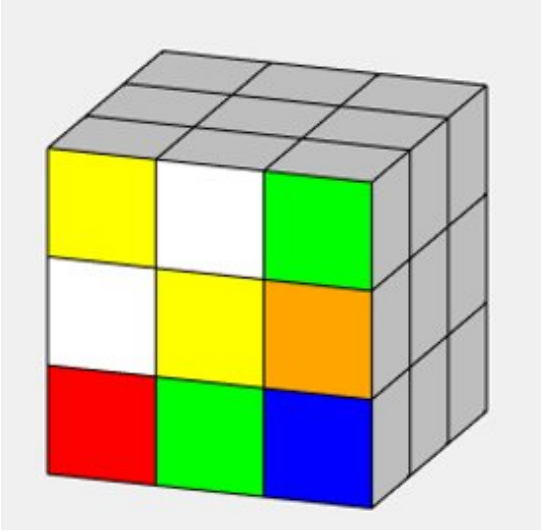

Figure 8: 3D visualization of the data from **Table 2**

Our system also has the capability to modify the hardcoded color values seen in **Table 1** dynamically. This can be done by showing a side of the cube and then pressing a number on the keyboard from 1-6 to change the corresponding color to be the color detected at the center sticker of the current face. This is important in making sure that our system is able to perform well in a variety of different lighting conditions. The effectiveness of this approach is discussed in our results. To make this clear to the user, we display the current six colors. Below is an example of both the default and modified color schemes. Although the difference is slight, it is clear that calibration made red, white, and yellow slightly lighter to match the lighting conditions.

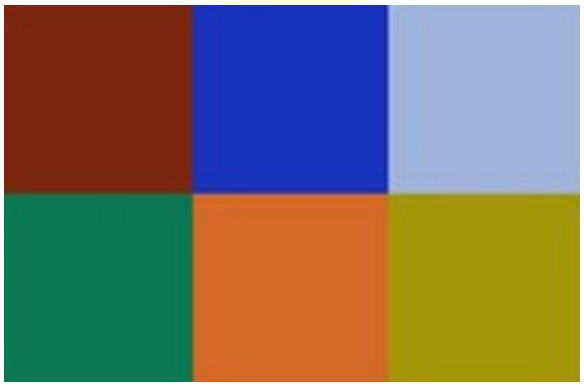

Figure 9: Original color scheme

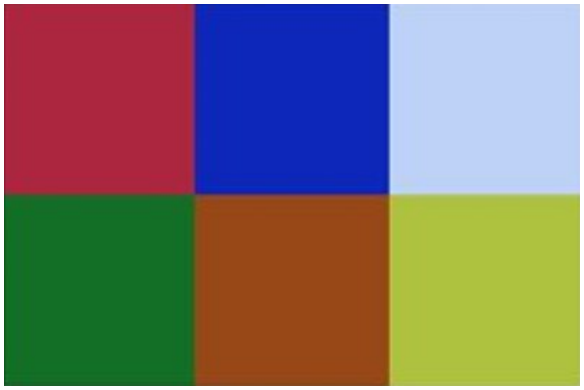

Figure 10: Color scheme after calibration

3.6 Getting Cube Representation from Six Faces

Next, our system allows the user to input all six faces of a Rubik's cube. This is done by using the arrow keys to cycle between the sides of the cube. Figure 11 and Figure 12 show the process of reading in two of the faces of a Rubik's cube.

Figure 11: Reading in the first side

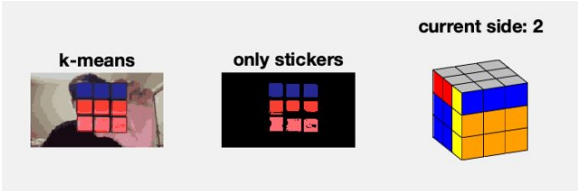

Figure 12: Reading in the second side

As seen in these images, our system automatically rotates the 3D model of the cube so the user can tell which side to scan next and verify that the scan was successful. The user then continues this process to do this for all six faces of the cube. Now, we have a full representation of the state of the cube.

3.7 Generate Solution

Finally, we must produce the solution for the identified state of the Rubik's cube. This work is done by the Solve45 function from [5]. This function uses the Thistlethwaite 45 algorithm, which is guaranteed to solve the cube in less than 45 moves, and it averages 31 moves per solution. This solution is output as an array of strings that look like U2 or L' which are in "Singmaster notation" [6]. This notation can be hard to read so we also visualize each move on the 3D model to make the moves easier to follow. The interface for this can be seen in the figure below. The top of the screen tells the next move. When the user presses any key, the move will be performed on the cube, and the window will wait for the user to press another key.

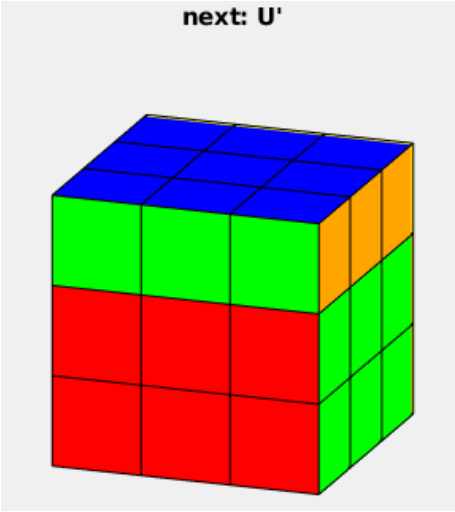

Figure 13: Interface for solving the cube

4. EXPERIMENTAL SETUP AND RESULTS

4.1 Experimental Setup

The experimental setup is as shown in **Figure 14**. It consists of a laptop with a webcam, a keyboard as a user input device, and a monitor to visualize the result of recognition. When the program is running, the webcam takes snapshots continuously and feeds the latest one into the recognizer whenever the recognizer finishes processing the last frame. A tester places a randomly shuffled Rubik's cube in front of the webcam. The monitor shows the results when nine stickers on a side are detected. Then, the tester manually compares results from our system to the ground truth and records the number of misclassified stickers. After that, the tester rotates the cube to an unrecognized face. For one cube, the process repeats until all six faces are recognized.

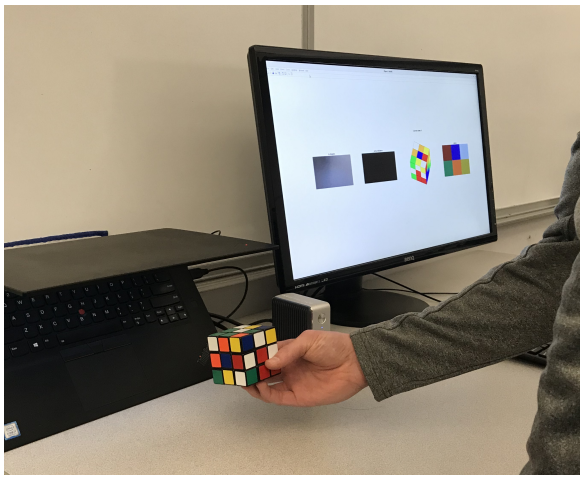

Figure 14: Experimental Setup for Rubik's Cube on a Clear Background

In our experiment, we perform an orthogonal array test which includes two factors - background and calibration - that potentially impact detecting accuracy. Each factor has two levels. **Figure 15** is an example of a simple "clean" background (top) and a "noisy" background (bottom), respectively. Each cube is recognized twice to capture the calibration effect: first with preset sticker colors and then with calibrated sticker colors (as described in **Section 3.5**).

Figure 15: Example Background of Experiment (top: "clean" background, bottom: "noisy" background)

The accuracy is defined as the proportion of correctly recognized stickers among all stickers detected. For example, if a system misclassified 5 stickers on a cube, the accuracy for this cube is:

$$Accuracy = \frac{6 \times 9 - 5}{6 \times 9} \times 100\% = 90.7\%$$

4.2 Results

Table 3 summarizes the average accuracy of our system under four different conditions (combination of two factors). The Rubik's cube detector achieved 89% and 96% accuracy, respectively, on the noisy and clear backgrounds without any calibration. This means that our system falsely predicts 6 and 2 stickers for a single cube on average. In these failures, the detector is mostly confused between blue/green, red/orange, and white/yellow sticker pairs depending on the lighting of different environments. With the color calibration for a specific light condition, the overall accuracy increases to 98% and 100%. The system is expected to obtain one false recognition on a "noisy" background and detect everything correctly on a "clean" background.

		Background	
		Noisy	Clean
Calibration	No	89%	96%
	Yes	98%	100%

Table 3: Accuracy on recognizing stickers on a Rubik's cube without calibration (cube 1 with a noisy background and cube 2 with a clean one)

5. DISCUSSION

Our system was overall accurate enough to allow for a Rubik's to be solved in a reasonable amount of time. This process greatly relies on the ability to calibrate the colors on the cube. There are some limitations to the system, however.

First, the system is affected in a couple of different ways by non-optimal lighting conditions. The below figure shows an example of slight reflections showing up in the bottom two blue stickers. This was probably caused by the bright laptop screen that the cube was close to. This made the K-Means clustering separate these bottom two blue stickers into two distinct colors. This is not ideal, and it can be seen from the sticker mask that this makes the view of the face not as clear. This is remedied in the actual system by simply tilting the cube slightly up or down to get slightly lighting.

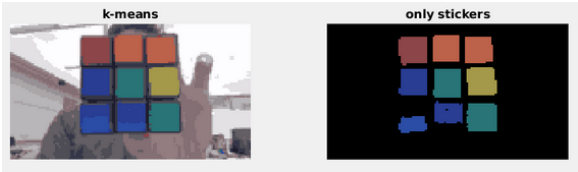

Figure 16: Example of reflection splitting stickers into multiple K-Means clusters

Next, there is the problem of guessing the wrong color for a specific sticker. In the below figure, the system thinks that the two orange stickers are actually red. This is before doing any sort of calibration, so the color it sees for orange is dark enough in the image that it appears to be closer to the default red value than the orange value. This can sometimes be

fixed by tilting the cube slightly, getting slightly different lighting to see if the orange color can be detected. The best method to fix this, however, would just be to perform calibration for red and orange.

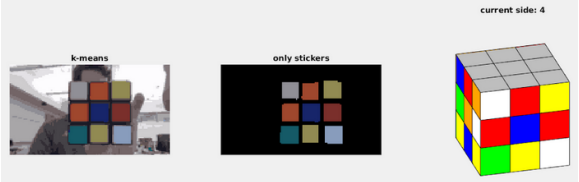

Figure 17: Example of incorrectly identifying colors before calibration

From both test cubes discussed in Section 4 that used calibration, there was only one sticker that was identified incorrectly. This can be seen in the figure below. This error occurred with the noisy background. It incorrectly labeled the top-left red sticker as orange. This is the same kind of issue we saw above without calibration, but calibration made this happen much less. Red and orange do appear to humans to be the hardest colors to differentiate, so it is expected that this might also prove difficult for our system.

Figure 18: Only incorrect prediction after calibration

Despite these inaccuracies, our system was able to perform relatively well after calibration was applied. From all 108 stickers seen, only one was labeled as the wrong color. An example of one side of the calibrated, clean background is shown below. The sticker mask clearly picks out each sticker and the colors are correctly identified.

Figure 19: Correct prediction with calibration

Because of how this system is used, it is perfectly usable given this level of accuracy. The system has many chances to try to get the right answer because input is being taken from the webcam at about ten times per second. This means that one mistake can be fixed in the next frame. Because of this ability to quickly re-predict after a mistake, this system provides a quick way to solve a Rubik's cube.

6. CONCLUSIONS AND FUTURE WORK

We were able to substantially improve the accuracy of the system by first calibrating it to the specific lighting conditions. This allows the system to know what exact RGB values correspond to which stickers. This also allows the system to represent stickers that are not the same colors as the standard set of six colors.

One of the main limitations of the current system is the restrictions on rotation of the image. Due to how the squareness is calculated when creating the sticker masks, a sticker that is not parallel to the horizontal axis will not be registered as a sticker. While this is simple to enforce for the user, a more robust system would be able to recognize the stickers as square, even if they are rotated. One possible way to do this would be to use Hough feature extraction to recognize squares of a size within a certain length within the image, and use the recognized squares as a basis for creating the sticker masks. However, this would likely lower the performance of the overall system.

Another possible improvement would be to allow the system to recognize multiple sides from a single image. Currently, the user needs to take six distinct images in order to recognize a full cube. With more time, it may be possible to allow the system to take a picture with three sides of the cube visible, and use that to identify all three sides at once. However, this would require substantial changes to how the sticker masks are produced, as the stickers in the image would no longer be square, and so any squareness measure would fail to detect the stickers.

7. KEY CHALLENGES AND LESSONS LEARNED

One initial issue that we encountered was the lack of a standardized training set. In our research, we found several sets of training images, but none of them were annotated, and so we were unable to use these for our system. Because of this, most of the testing done when creating the system was done manually by

using the webcams on the team's laptops. This allowed us to easily test the system's performance without the need for a full testing set.

We also faced challenges with how to take in live input from the user's webcam. When taking input, we encountered issues with updating the figures on screen while still allowing the users to interact with them and reading input. Setting the figures to update on a timer prevented the user from interacting with the figures in any way, particularly the 3D model of the cube, which led to difficulty when checking to see if an image was read properly. To combat this, we added a user input that paused the feed from the webcam. This meant that if the user wanted to look at other sides of the cube model, they could simply pause input, and rotate the display as needed. This also meant that the other images on screen could be updated live as input was processed from the webcam.

Another issue that we faced was that of performance. In our initial versions of the system, processing a single frame could take upwards of six seconds. For most systems, this is a relatively small amount of time, but for a system that requires live input from the user's webcam, this six second delay caused major issues. We were able to substantially reduce the computation time required for an image by downscaling the image by a factor of ten. This scale allowed the system to run much faster, and seemed to be the smallest the images could be made before the system began to have trouble recognizing the individual squars of the cubes.

Finally, we had some issues with recognizing the cube stickers in different lighting conditions. We had tested the system under one set of conditions, getting the accuracy to an acceptable level, but when moving to a new space, the tuning of the color thresholds no longer gave us a good enough level of accuracy. We solved this issue by allowing the user to calibrate the color thresholds as needed, which is discussed in more detail above.

8. REFERENCES

[1] J. Hack and K. Shutzberg, "Rubik's Cube Localization, Face Detection, and Interactive Solving", *cs231n.stanford.edu*.

[2] Włodzimierz Kasprzak, Wojciech Szynkiewicz, and Lukasz Czajka. Rubik's Cube Reconstruction from Single View for Service Robots. Warsaw University of Technology.

[3] Barthelet, Luc. *Introduction to Image Processing: Solving the Rubik's Cube from Pictures -- from Wolfram Library Archive*, Wolfram Research, 2012, https://library.wolfram.com/infocenter/Conferences/8333/.

[4] Feng, Lin, et al. "Color Recognition for Rubik's Cube Robot." *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, 2019, https://doi.org/10.1109/smartiot.2019.00048.

[5] Heit, Joren. "Rubik's Cube Simulator and Solver." *MathWorks*, 23 Oct. 2011, https://www.mathworks.com/matlabcentral/fileexchange/31672-rubik-s-cube-simulator-and-solver.

[6] "Rubik's Cube Move Notation." *Wikipedia*, Wikimedia Foundation, 11 Feb. 2022, https://en.wikipedia.org/wiki/Rubik%27s_Cube#Move_notation.