GROUP THEORY IN THE RUBIK'S CUBE

MICKY SANTIAGO-ZAYAS

ABSTRACT. This project will focus on defining notation and expressing with group theory mathematics of the Rubik's cube. Starting off with a proof of why the Rubik's cube rotations form a group. To later define all the elements of the group as cycle notations. Additionally, evaluate what are the possible configurations that a cube may have, which would help us determine in the program a legal randomized starting configuration. Then, would like to explain a notation to define orientation and position and how these can help figure out a method to solve the cube. With it, we can dive into theoretical considerations of what the Rubik's group is, namely, isomorphic groups. Finally, would use a code in Python, to determine from the initial state of the cube if it is a valid configuration or not.

1. INTRODUCTION

A well-known combination puzzle was invented in 1974 by Ernö Rubik, consisting of a cube with 27 smaller cubes (called cubies) of 6 different colors. Its solved configuration occurs when all the cube's faces are of the same color. It has inspired world championships to declare the fastest person to solve it (the current world record for solving the cube is 5.55 seconds.) In addition, it has inspired many great studies to understand the puzzle. Finding ways to solve the most efficient way possible and determining what the minimum number of moves from the solved state to the furthest configuration from it is. This number is called God's Number and in 2010 was determined to be 20 [4]. The challenge is to find a way to solve an unsolved configuration the most efficient way or the least moves possible.

The cube can be expressed with Group Theory notation. Where the different transformations and configurations of the cube form a subgroup of a permutation group generated by the horizontal and vertical rotations of the puzzle. The solution to the cube can also be described by Group Theory. Group Theory allows for the examination of how the cube functions and how the twists and turns return the cube to its solved state. [4]

2. CUBE NOTATION

2.1. **Basic Principles.** We use a notation developed by David Singmaster that is incredibly common, where we refer to the faces as Right (r), Left (l), Up (u), Down (d), Front (f), and Back (b). When we are referring to the moves of a given face as a one clockwise 90° turn, we will write them as uppercase letters of the corresponding face, R, L, U, D, F, and B.

Definition 1. As defined by [3], *Cubie and Cubicle*: A cubie is one of the 26 colored blocks on the Rubik's cube. A corner cubie has 3 visible faces while an edge cubie only has 2. When referring to the cubies we will name them based off of the starting location of the cubie, not on the colors of the faces. A Cubicle, on the other hand, is the space in which the cubie lives. If you rotate the face of the cube the cubicles do not move but the cubies do.

Date: April 30, 2022.

More specifically, to name a corner or edge cubie we list the visible faces in clockwise order. For example the cubie that lies on the up, right, front corner of the cube is named urf. This cubie can also be referred to a rfu or fur if we do not care about the orientation of the cubie.

Definition 2. *Orientation*: The orientation of a cubie refers to the position the cubie has been twisted into no matter what cubicle the cubie is in. Thus the urf cubie can be in the urf cubicle but can be in the orientation of fur because the original Front face is on the Up face, the Up on the Right, and the Right on the Front. Thus, when we are referring to oriented cubies rfu, fur, and urf are not the same. The notation for orientation will be disccussed in section 2.4.

2.2. **Cycle Notation.** If you look at rfu after the move R is applied to it, it moves to the position rub. Meanwhile rub moves to rbd, rbd moves to rdf and rdf moves to rfu. We can write these moves, combined with what the move R does to the edge pieces, in cyclic notation as, $R = (rfu, rub, rbd, rdf)(ru, rb, rd, rf)$ [3]. As a consequence, we can se each of the elements of $\mathbb{G}$ as disjoint cycles,

$$D = (dlf, dfr, drb, dbl)(df, dr, db, dl) \tag{2.1}$$

$$R = (rfu, rub, rbd, rdf)(ru, rb, rd, rf) \tag{2.2}$$

$$U = (ulb, ubr, urf, ufl)(ul, ub, ur, uf) \tag{2.3}$$

$$L = (luf, lfd, ldb, lbu)(lu, lf, ld, lb) \tag{2.4}$$

$$F = (fur, frd, fdl, flu)(fu, fr, fd, fl) \tag{2.5}$$

$$B = (bul, bld, bdr, bru)(bu, bl, bd, br) \tag{2.6}$$

[3]

2.3. **Position.** We are going to label each edge and corner cubie with a number as done by Cooke and other studies as follows:

For the corner cubies

1 on the u face of the ufl cubie
2 on the u face of the urf cubie
3 on the u face of the ubr cubie
4 on the u face of the ulb cubie
5 on the d face of the dbl cubie
6 on the d face of the dlf cubie
7 on the d face of the dfr cubie
8 on the d face of the drb cubie

For the edge cubies

1 on the u face of the ub cubie
2 on the u face of the ur cubie
3 on the u face of the uf cubie
4 on the u face of the ul cubie
5 on the b face of the lb cubie
6 on the b face of the rb cubie
7 on the f face of the rf cubie
8 on the f face of the lf cubie
9 on the d face of the db cubie
10 on the d face of the dr cubie

11 on the d face of the df cubie
12 on the d face of the dl cubie
[3]

For the corner cubies we can write this as an element σ of S_8 (the element of S_8 which moves the corner cubies from their starting positions to the new positions) where $\sigma : \{1, 2, 3, 4, 5, 6, 7, 8\} \to \{1, 2, 3, 4, 5, 6, 7, 8\}$. Thus σ is defined by the placement of the corner cubies in their resulting cubicles after a specific move has been performed. For the edge cubies we can write this as an element of τ of S_{12} where $\tau : \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \to \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$. Thus τ is defined by the placement of the edge cubies in their resulting cubicles after a specific move has been performed. [3]

2.4. **Orientation.** We must also assess the notation for the orientation of the cubies as we have not addressed the problem between fur, urf, and ruf. Simply follow the notation from this image in [**?**, p. 15]daniels

2.3. Cube Position. From Lemma 4.3 any corner cube position can be expressed as a 8-tuple and from Lemma 4.4 any edge cube position can be expressed as a 12-tuple. However, to determine the individual components of the tuples, a fixed numbering system will be needed.

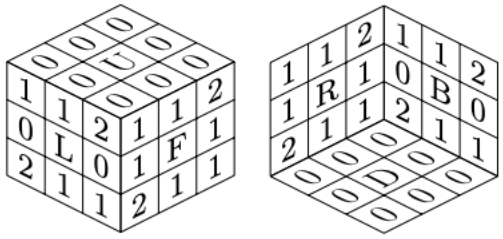

FIGURE 4. The fixed orientation markings, as denoted in [**5**], for the facets of the Rubik's Cube [**6**].

3. RUBIK'S CUBE

3.1. **The Rubik's Cube Group.**
On the Rubik's Cube, there are 54 facets that can be arranged and rearranged through twisting and turning the faces. Any position of the cube can be describe as a permutation from the solved state. Thus, the Rubik's Cube group is a subgroup of a permutation group of 54 elements. [4, p. 13]

Definition 3. The permutation group $\mathbb{G} = \{F, L, U, D, R, B\} \subset S_{54}$ is called the Rubik's Cube Group.[4]

Theorem 3.1. *The set of moves of the Rubik's Cube is a group denoted* $(\mathbb{G}, *)$.

Proof. In order to show that ($\mathbb{G}$, *) is a group we must show that $\mathbb{G}$ closed under $*$, that a right identity exists, a right inverse exists, and $*$ is associative. Let M_1 and M_2 be two moves in $\mathbb{G}$. If M_1 and M_2 are moves then consider $M_1 * M_2$ is a move as well. Thus $\mathbb{G}$ is closed under $*$.
If we let e be the "empty move", which means it does not change the faces of the Rubik's cube at all, then $M * e = M$. Hence $\mathbb{G}$ has a right identity.
If M is an arbitrary move, then let the reverse steps of that move be M_0. Then $M * M_0$ means to first do all the moves of M and then undo all of the moves of M, leaving us in the same configuration

we started in. Thus $M * M_0 = e$ and therefore M_0 is the inverse of M and every element of $\mathbb{G}$ has a right inverse.
If C is an oriented cubie, we will write $M(C)$ for the oriented cubicle that C ends up in after we apply the move M, with the faces of $M(C)$ written in the same order as the faces of C. That is, the first face of C should end up in the first face of $M(C)$, and so on. Let M_1 and M_2 be two moves in G, then $M_1 * M_2$ is the move where we first do M_1 and then do M_2. The move M_1 moves C to the cubicle $M_1(C)$ and the move M_2 then moves it to $M_2(M_1(C))$. Thus, $(M_2 * M_1)(C) = M_2(M_1(C))$. To show $*$ is associative, we must show that $(M_1 * M_2) * M_3 = M_1 * (M_2 * M_3)$ for any moves M_1, M_2, and M_3. That is, we want to show that $[(M_1 * M_2) * M_3](C) = [M_1 * (M_2 * M_3)](C)$ for any cubie C. As previously stated in this proof we know that $[(M_1 * M_2) * M_3](C) = M_3([M_1 * M_2](C)) = M_3(M_2(M_1(C)))$. On the other hand, $[M_1 * (M_2 * M_3)](C) = (M_2 * M_3)(M_1(C)) = M_3(M_2(M_1(C)))$. So, $(M_1 * M_2) * M_3 = M_1 * (M_2 * M_3)$. Thus * is associative. Hence, $(\mathbb{G}, *)$ is a group.[3] □

Theorem 3.2. $\mathbb{G}$ *is not abelian.*

Proof. Let M_1 be R and M_2 be B. Let us look at cubie urf in starting position. M_1M_2 moves urf to the ulb position. Now if we do moves M_2M_1 with urf in the starting position the urf cubie moves to the bru position. Since M_1M_2 and M_2M_1 produce different configurations, which we see from urf being in different positions, we can say that $\mathbb{G}$ is not abelian. [3] □

Theorem 3.3. *Let C_1 and C_2 be two different unoriented corner cubies, there is a move of the Rubik's cube which sends C_1 to C_1' and C_2 to C_2'.*

Proof. Let this be a proof by cases. Let C_1 and C_2 be two different unoriented corner cubies. Suppose that C_1 and C_1' share a face, without loss of generality call it f. Then F^n send C_1 to C_1'. Let C_2"denote the position of C_2 after F^n. If $C_2" = C_2'$ then we are done.
Suppose $C_2" \neq C_2'$. Since $C_1 \neq C_2$, we can't have $C_1' = C_2'$ or else two different cubies would occupy the same position. So C_1', C_2', and C_2" are three different corner cubies.
Case 1 There exists a face, shared by C_2" and C_2' but not C_1'. Without loss of generality call it b, rotate it B^n times and we have $C_1 \to C_1'$ and $C_2 \to C_2'$.
Case 2 C_2" and C_2' share a face with C_1'.
Case a More than 2 faces are shared with C_2", C_2', and C_1'. This is impossible based on the configuration of the Rubik's Cube.
Case b C_2" and C_2' do not share a face. Since every corner cubie shares a face with all but one other corner cubie, C_2" and C_2' each share at least one face with C_1'. C_2" and C_1' do not share at least one face, without loss of generality, call that face l. C_2" can be moved L^n times to share a face with C_2' but not share a face with C_1'. Without loss of generality call this face R, then C_2" can be moved to C_2' by R^n moves. Thus, $C_1 \to C_1'$ and $C_2 \to C_2'$.
Suppose that C_1 and C_1' do not share a face. Note, each corner cubie does not share a face with only one other corner cubie. Thus, C_1 can be moved by one move to any other adjacent corner and will now share a face with C_1', without loss of generality, call that face f. F^n moves will then put the C_1 cubie into the C_1' position. If $C_2" = C_2'$ then we are done.
Suppose $C_2" \neq C_2'$. Note, as mentioned earlier $C_1 = C_2$ and $C_1' \neq C_2'$. So C_1', C_2', and C_2' are three different cubies.
Case 1 There exists a face shared by C_2" and C_2' but not C_1'. Without loss of generality call this face u. U^n can move C_2" and C_2' and therefore, $C_1 \to C_1'$ and $C_2 \to C_2'$.
Case 2 There does not exist a face shared by C_2" and C_2 but not C_1.

Case a Every face shared by C_2" and C_2' is shared with C_1'. This is impossible based on the configuration of the Rubik's Cube.
Case b C_2" and C_2' do not share a face. Since every corner cubie shares a face with all but 1 other cubie, C_2" and C_2 both share at least one face with C_1'. There is a face on C_2" that does not share a face with C_1' (otherwise they would be in the same position). Without loss of generality call that face d. Rotate C_2". D^n times such that C_2" and C_2' share a face (which is possible since only the original position of C_2" will not share a face with C_2').
C_2" does not share the same face that C_2' and C_1 share. Without loss of generality, call the face that C_2" and C_2', but not C_1, share r. Rotate the face R^n times until C_2" is in the C_2' position. Hence, $C_1 \to C_1'$ and $C_2 \to C_2'$.
Thus, there is a move of the Rubik's cube which sends C_1 to C_1' and C_2 to C_2'. [3] □

3.2. **Valid Configurations.**

Notice, that although, theoretically there are 8 corner cubies there are $8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 8!$ possible positions of the corner cubies. Since their are 3 faces of a corner cubie, and therefore 3 orientations of a corner cubie, there are 3^8 possible ways the corner cubies could be oriented. In the same way we can look at the edge cubies and see their are 12! possible positions and with 2 different faces 2^{12} different orientations. Thus, if we combine all of the different possibilities we see that $2^{12}3^88!12!$ or 5.19×10^{20} that is about 519 quintillion potential configurations [3]. They form part of an illegal Rubik's Cube Group.

Definition 4. The Illegal Rubik's Cube Group allows the solver to take the cube apart and reassemble it in any orientation. Again, some of the orientations are not physically possible on the cube. When all the possible positions of the facets are combined as a whole, some of the arrangements will not be physically possible on the cube. [4, p. 16]

However, as mentioned before, not all these configurations are physically actually possible. There are a set of conditions required so that a specific configuration to be possible from the solved state. First, let us define and show some specifics. Define the polynomial

$$\Delta = \prod_{1 \le i < j \le n} (x_i - x_j). \tag{3.1}$$

Theorem 3.4. *For any $\sigma \in S_n$, $\Delta\sigma = \pm\Delta$.*

Proof. By definition, Equation (3.1) so

$$\Delta\sigma = \prod_{1 \le i < j \le n} (x_{\sigma(i)} - x_{\sigma(j)}). \tag{3.2}$$

In order to show $\Delta\sigma = \pm\Delta$, we must show two things. First, for each i and j with $1 \le i < j \le n$, we must show that either $x_{\sigma(i)} - x_{\sigma(j)}$ or its negative appears in Δ; that is, either $x_{\sigma(i)} - x_{\sigma(j)}$ or its negative has the form $x_k - x_l$ with $1 \le k < l \le n$. Secondly, we must show that, for each i and j with $1 \le i < j \le n$, either $x_i - x_j$ or its negative appears in $\Delta\sigma$. Since Δ and $\Delta\sigma$ have the same number of terms, these two statements together prove that the terms of Δ and $\Delta\sigma$ match up.
To prove the first statement, all we need to show is that either $\sigma(i) < \sigma(j)$ or $\sigma(j) < \sigma(i)$; equivalently, we need to show that $\sigma(i) \ne \sigma(j)$ if $1 \le i < j \le n$. This is true because σ is one-to-one and $i \ne j$.
To prove the second statement, we need to show that either $x_i - x_j$ or its negative can be written as $x_{\sigma(k)} - x_{\sigma(l)}$ with $1 \le k < l \le n$. Since $\sigma \in S_n$, $\sigma^{-1} \in S_n$; in particular, σ^{-1} is also a bijection.

Since $i \neq j$, $\sigma^{-1}(i) \neq \sigma^{-1}(j)$. Let k be the smaller of $\sigma^{-1}(i)$ and $\sigma^{-1}(j)$, and let l be the larger. Then, $1 \leq k < l \leq n$, and $x_i - x_j$ is either $x_{\sigma(k)} - x_{\sigma(l)}$ or its negative. [1, p. 28] □

With the theorem and definitions above we can then state that the sign of σ, denoted $sgn\sigma$ is equal to the sign of Δ when calculating $\Delta\sigma$.

Theorem 3.5. *A configuration (σ, τ, x, y) is valid if and only if the $sgn\sigma = sgn\tau$ (equal parity of permutations), $\sum x_i \equiv 0 \mod 3$ (conservation of the total number of twists), and $\sum y_i \equiv 0 \mod 2$ (conservation of the total number of flips).* [4, 1]

Theorem 3.6. *If (σ, τ, x, y) and (σ', τ', x', y') are in the same orbit, then $(sgn\sigma)(sgn\tau) = (sgn\sigma')(sgn\tau)$.* [1]

Proof. By [1, Lemma 10.12], it suffices to show that, if $(\sigma', \tau', x', y') = (\sigma, \tau, x, y) \cdot M$ where M is one of the 6 basic moves, then $(sgn\sigma)(sgn\tau) = (sgn\sigma')(sgn\tau')$. By [1, p. 28, problem 3], $\sigma' = \sigma\phi_{\text{corner}}(M)$ and $\tau' = \tau\phi_{\text{edge}}(M)$. Therefore, $(sgn\sigma')(sgn\tau') = (sgn\sigma)(sgn\phi_{\text{corner}}(M))(sgn\tau)(sgn\phi_{\text{edge}}(M))$. If M is one of the 6 basic moves, then $\phi_{\text{corner}}(M)$ and $\phi_{\text{edge}}(M)$ are both 4-cycles, so they both have sign -1. Thus, $(sgn\sigma')(sgn\tau') = (sgn\sigma)(sgn\tau)$. [1] □

Theorem 3.7. *If (σ, τ, x, y) is a valid configuration, then $sgn\sigma = sgn\tau$.* [1]

Proof. This is a direct consequence of Theorem 3.6 since any valid configuration is in the orbit of the start configuration (1, 1, 0, 0). [1] □

Theorem 3.8. *If (σ', τ', x', y') is in the same orbit as (σ, τ, x, y), then $\sum x_i' \equiv \sum x_i \mod 3$ and $\sum y_i' \equiv \sum y_i \mod 2$.* [1]

The details of this proof are in page 35 of [1]. As a corollary, we can find.

Theorem 3.9. *If (σ, τ, x, y) is a valid configuration, then $\sum x_i \mod 3$ and $\sum y_i \mod 2$.* [1, Corollary 11.5]

The details to this proof are in page 36 of Chen's paper [1]. To complete the proof of theorem 3.5, we are left to show these following theorems from [1]. Proofs and details follow from other lemmas in [1] that can be found in pages 37-39.

Theorem 3.10. *If (σ, τ, x, y) is a configuration such that $sgn\sigma = sgn\tau$, $\sum x_i \equiv 0 \mod 3$, and $\sum y_i \equiv 0 \mod 2$, then the orbit of (σ, τ, x, y) contains some configuration of the form $(1, \tau', x', y')$.*

Theorem 3.11. *If $(1, \tau, x, y)$ is a configuration with $sgn\tau = 1$, $\sum x_i \equiv 0 \mod 3$, and $\sum y_i \equiv 0 \mod 2$, then the orbit of $(1, \tau, x, y)$ contains some configuration of the form $(1, \tau', 0, y')$.*

Theorem 3.12. *If $(1, \tau, 0, y)$ is a configuration with $sgn\tau = 1$ and $\sum y_i \equiv 0 \mod 2$, then the orbit of $(1, \tau, 0, y)$ contains some configuration of the form $(1, 1, 0, y)$.*

Theorem 3.13. *If $(1, 1, 0, y)$ is a configuration with $\sum y_i \equiv 0 \mod 2$, then the orbit of $(1, 1, 0, y)$ contains the start configuration $(1, 1, 0, 0)$.*

4. Solving the Rubik's Cube

4.1. **Top Layer.** First we are going to solve the top layer. For that we need to make sure that the edge pieces of the cross align with the colors of the middle pieces on all the faces surrounding it. After the cross is formed, the corner pieces that match the colors need to be put into the right

position, as well as orientated correctly. We will do so by cases. First, try to position ub cubicle in 1. By the following cases:

$$2 - U^{-1}\ 3 - uf : U^2\ 4 - ul : U\ 5 - lb : B^{-1}$$

$$6 - rb : B\ 7 - rf : RU^{-1}\ 8 - lf : L^{-1}U$$

$$9 - db : B^2\ 10 - dr : DB^2\ 11 - df : D^2B^2$$

$$12 - dl : D^{-1}B^2$$

To change its orientation, we must just apply $B^{-1}R^{-1}U^{-1}$. Having one cubicle in position and orientation, now turn the whole Rubik's Cube such that the cubie we just placed is now the ul cubicle and the cubicle we are trying to solve next is the ub cubicle. Do not worry about orientation, we are just trying to get the cubie into the correct position. Again, by the following cases:

$$2 - uf : RB\ 3 - ul : FR^2B\ 5 - lb : B^{-1}$$

$$6 - rb : B\ 7 - rf : R^2B\ 8 - lf : F^{-1}D^2B^2$$

$$9 - db : B^2\ 10 - dr : DB^2\ 11 - df : D^2B^2$$

$$12 - dl : D^{-1}B^2$$

The move we will use to change the orientation is $B^2D^{-1}R^{-1}B$. For the third edge, hold the Rubik's Cube such that the positioned edge pieces are in the ul and uf cubicles. We are going to be solving the ub cubicle. Apply from the following cases:

$$2 - uf : RB\ 5 - lb : B^{-1}\ 6 - rb : B$$

$$7 - rf : R^2B\ 8 - lf : F^{-1}D^2B^2F\ 9 - db : B^2\ 10 - dr : DB^2$$

$$11 - df : D^2B^2\ 12 - dl : D^{-1}B^2$$

To orient it correctly as done before, apply $B^2D^{-1}R^{-1}B$. Hold the Rubik's Cube such that the face without the correct cubie in the cubicle is the Front face, such that the cubicle we are trying to get the cubie in is uf. If in the front face, F or F^{-1} as needed to get into position. Otherwise, rotate the down face until the cubie is in the front (oriented or not), then rotate the front until in position (oriented or not). To orient, $FR^{-1}D^{-1}RF^2$. With the edge cubies forming a cross on the Up face we can look to place the correct corner cubies in the Up face cubicles. We will be using the move $DRD^{-1}R^{-1}$,or $[D, R]$.

If the cubicle is in the down face, hold the cubie such that the cubicle we need to move the cubie into is in the ubr cubicle, position 3 by the definition of σ, then twist the Down face either once, twice, once counter clockwise, or don't twist it at all, such that we end up with the cubie we are moving in the dbr cubicle. If the cubie we need to move is in position 5, dbl, then twist the Down face once counterclockwise, if the cubie is in position 6, dlf, then twist the Down face twice, if the cubie is in position 8, dfr, then twist the Down face once clockwise, and if the cubie is in position 8, dbr, then keep the cubie where it is. Then depending on the orientation of the cubie we will need to perform $[D, R]$ either once, twice, or five times. If in the Up face, apply $[D, R]$ to put the cubicle in the down face, and would apply the other case.

4.2. **Middle Layer.** Now, the plan is to work on the sides adjacent to the top, more specifically, its edges outside the down layer. The goal is to position and orient them correctly. If the desired cubie is in the down face, rotate it until its color aligns with the center of one of the faces. Having that face as the front, if the bottom color matches the one to the right, then apply $D^{-1}R^{-1}DR$. If the one to the left, $DLD^{-1}L^{-1}$. If all the edges that do not possess the bottom color are not in the bottom layer, get one of the unoriented edges to the right face, and apply $D^{-1}R^{-1}DR$. Then go back to case 1.

4.3. **Bottom Layer.** As in the Top-Layer, we wish to form a cross in the bottom face. No edge cubies in the right orientation, one edge cubie in the right orientation, two edge cubies in the right orientation on opposite faces, two edge cubies in the right orientation on adjacent faces, or all four cubies in the right orientation. In case 1, apply $RDFD^{-1}F^{-1}R^{-1}$ which is $R[DF]R^{-1}$ to get one of the other cases. In the second one, hold the Rubik's Cube during the such that the edge cubies that match the middle color are in the df and dl cubicles. The third case, perform the move in this position and be sure to hold the cube such that the edge cubies that match the color of the middle Down face cubie are in the dl and db cubicles. All this is for the desired fourth configuration.

Now that the Down face edge cubies have the right orientation, we have to get them into the right position. First rotate the Down face until at least one face has an edge cubie lined up with the same color on one of the side faces. Note from an earlier exercise that D does not affect the orientations of any of the edge cubies, only the positions of the Down face edge and corner cubies. Once you have an edge cubie lined up with its side color call this face F, such that the lined up edge cubie is in the df cubicle. Note, if you already have two adjacent faces where the edge cubies lined up you can skip this step. With your positioned cubie on the front face perform the move $F^{-1}D{-}1FD^{-1}F^{-1}(D^{-1})^2FD^{-1}D^{-1}$. From that move, we should now have two adjacent sides in the right position as well as the right orientation. When this is the case rotate your entire Rubik's Cube so one aligned edge is on the Back face, in the db cubicle, and the other is on the Left face, in the dl cubicle. Then, with the edge cubies that are in the wrong position on the Right and Left faces, perform the move $D^{-1}R^{-1}D^{-1}RD^{-1}R^{-1}(D^{-1})^2R$.

Next we can begin to look at the corner cubies on the Down face. If all four corners are in the wrong positions and orientations an extra step must be taken. The move $R^{-1}DLD^{-1}RDL^{-1}D^{-1}$ must be performed if all four corners cubies are in the wrong positions. From this point forward, in order to minimize potential mistakes, flip the Rubik's Cube so the bottom layer, that still has unoriented cubies, is now the Up face, and the other two solved layers are the bottom two layers. Once we have at least one corner in the correct orientation and position we need to look and see if the three remaining corner cubies are in the correct positions. If not in position, to get the other cubies into their correct cubicles we will perform the move $URU^{-1}L^{-1}UR^{-1}U^{-1}L$ with the already positioned cubie starting in the urf or 2 corner cubie position. Now, ensure the cube is positioned having any unoriented corner to the top right of the front. Then apply $R^{-1}D^{-1}RD$ until that cubicle is oriented. If there are still any unoriented cubicles, apply U until you position it the same as before. Repeat until all the corners are oriented. Now, to finish, apply U until all the colors match with their faces.

5. Generalizing & Validating Algorithm To Solve The Cube

Throughout the paper, we have been mostly informing about past studies made over the Rubik's Cube and its connection to Group Theory. Still, we can take it a step further and develop an algorithm than can manipulate and visual Rubik's Cube with the notation and theory developed so far. In section 6.1, there is a python script I developed to work on all possible manipulations of the Rubik's Cube. Taking advantage of the cycle notation for all 6 possible movements over the puzzle. Then, section 6.2, it is the script responsible of verifying the given cube configuration is valid. Namely, determining if a configuration is an element in the Rubik's Group in `valid_cube()` validating each condition as stated in theorem 3.5. section 6.2 would also have owned the function responsible of solving the configuration of a given cube, `solve()` as discussed in . However, due to lack of resources, especially for the debugging, the script only validates a cube's configuration is possible just with the studied 6 permutations. Finally, section 6.3 is the script needed to have an interaction with the other two scripts through the terminal.

For our purposes, let us focus on section 6.2. In it, we have helper functions to determine the validity of a configuration. First, we need to ensure that the total number of cubies per color do not surpass 8. Then, we have to determine the sign of the cycles for both the edges and corners. The algorithm was greatly inspired by [2, Theorem 2.10, Example 2.9]. The first step was to determine the cycles for each, which was a list of values going from 0 to 7 inserted in the index that returned them. For example, edge 1 ended in position of edge 2, thus $[x_0, x_1, 1, x_3, \ldots, x_7]$. Harrell's work helped visualize the steps for the algorithm [5]. Then, we need to calculate the conservation for both the twists and turns. The notation for both of them was employed using Harrell's visual representations.[5]

Now, you are more than welcome to run the script over your computer and have your first virtual Rubik's Cube. Even more, you can have the configuration of a physical cube put onto the system and make the $UFLBRD$ moves over the terminal. You can even try to make up an imaginary configuration and would determine if the configuration is part of the Rubik's Group or not. Even though the system is still not able to solve the cube for you, you can try to do it yourself with the virtual moves.

6. Appendix

6.1. **Code: Cube.py.**

```
import random

def generate_cube():
    # the top face is white
    '''
    Arrays are generated by the colors of the cubies
    with the whit face on top and red in the front.
    The color arrangement is the standard opposite faces
    red-orange, white-yellow, green-blue
    '''
    white = ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W']
    red = ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R']
```

```
    green = ['G', 'G', 'G', 'G', 'G', 'G', 'G', 'G']
    blue = ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B']
    yellow = ['Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y']
    orange = ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']

    '''DEBUGGING PURPOSES'''
    # white = [1, 2, 3, 4, 5, 6, 7, 8]
    # red = [1, 2, 3, 4, 5, 6, 7, 8]
    # green = [1, 2, 3, 4, 5, 6, 7, 8]
    # blue = [1, 2, 3, 4, 5, 6, 7, 8]
    # yellow = [1, 2, 3, 4, 5, 6, 7, 8]
    # orange = [1, 2, 3, 4, 5, 6, 7, 8]

    cube = object_cube(white, red, green, blue, yellow, orange)
    return cube

def is_solved(cube):
    (white, red, green, blue, yellow, orange) = ungroup_cube(cube)
    for i in range(0, 6):
        if (i == 0):
            arr = white
            value = 'W'
        if (i == 1):
            arr = red
            value = 'R'
        if (i == 2):
            arr = green
            value = 'G'
        if (i == 3):
            arr = blue
            value = 'B'
        if (i == 4):
            arr = yellow
            value = 'Y'
        if (i == 5):
            arr = orange
            value = 'O'
        for j in range(0, 8):
            if (arr[j] != value):
                return False
    return True

def print_cube(cube):
    (white, red, green, blue, yellow, orange) = ungroup_cube(cube)
    arr = []
    for i in range(0, 6):
```

```
        if (i == 0):
            arr = white.copy()
            array_name = "white_face"
        elif (i == 1):
            arr = red.copy()
            array_name = "red_face"
        elif (i == 2):
            arr = green.copy()
            array_name = "green_face"
        elif (i == 3):
            arr = orange.copy()
            array_name = "orange_face"
        elif (i == 4):
            arr = blue.copy()
            array_name = "blue_face"
        else:
            arr = yellow.copy()
            array_name = "yellow_face"
        print(f"{array_name}: {arr[0]}, {arr[1]},
        {arr[2]}, {arr[3]}, {arr[4]}, {arr[5]}, {arr[6]},
        {arr[7]}")

def rotate(face):
    return [face[6], face[7], face[0], face[1], face[2],
    face[3], face[4], face[5]]

def up(white, red, green, blue, yellow, orange):
    placeholder = [blue[0], blue[1], blue[2]]
    # replace orange to blue
    blue[0] = orange[0]
    blue[1] = orange[1]
    blue[2] = orange[2]
    # replace green to orange
    orange[0] = green[0]
    orange[1] = green[1]
    orange[2] = green[2]
    # replace red to green
    green[0] = red[0]
    green[1] = red[1]
    green[2] = red[2]
    # replace blue to red
    red[0] = placeholder[0]
    red[1] = placeholder[1]
    red[2] = placeholder[2]
    #rotate white
    white = rotate(white)
```

```
    return white, red, green, blue, yellow, orange

def right(white, red, green, blue, yellow, orange):
    placeholder = [yellow[2], yellow[3], yellow[4]]
    # replace orange to yellow
    yellow[2] = orange[6]
    yellow[3] = orange[7]
    yellow[4] = orange[0]
    # replace white to orange
    orange[6] = white[2]
    orange[7] = white[3]
    orange[0] = white[4]
    # replace red to white
    white[2] = red[2]
    white[3] = red[3]
    white[4] = red[4]
    # replace yellow to red
    red[2] = placeholder[0]
    red[3] = placeholder[1]
    red[4] = placeholder[2]
    #rotate blue
    blue = rotate(blue)
    return white, red, green, blue, yellow, orange

def left(white, red, green, blue, yellow, orange):
    placeholder = [white[0], white[6], white[7]]
    # replace orange to white
    white[0] = orange[4]
    white[7] = orange[3]
    white[6] = orange[2]
    # replace yellow to orange
    orange[2] = yellow[6]
    orange[3] = yellow[7]
    orange[4] = yellow[0]
    # replace red to yellow
    yellow[0] = red[0]
    yellow[7] = red[7]
    yellow[6] = red[6]
    # replace white to red
    red[0] = placeholder[0]
    red[6] = placeholder[1]
    red[7] = placeholder[2]
    #rotate green
    green = rotate(green)
    return white, red, green, blue, yellow, orange
```

```
def back(white, red, green, blue, yellow, orange):
    placeholder = [white[0], white[1], white[2]]
    # replace blue to white
    white[0] = blue[2]
    white[1] = blue[3]
    white[2] = blue[4]
    # replace yellow to blue
    blue[2] = yellow[4]
    blue[3] = yellow[5]
    blue[4] = yellow[6]
    # replace green to yellow
    yellow[4] = green[6]
    yellow[5] = green[7]
    yellow[6] = green[0]
    # replace white to green
    green[6] = placeholder[0]
    green[7] = placeholder[1]
    green[0] = placeholder[2]
    #rotate orange
    orange = rotate(orange)
    return white, red, green, blue, yellow, orange

def down(white, red, green, blue, yellow, orange):
    placeholder = [green[4], green[5], green[6]]
    # replace orange to green
    green[4] = orange[4]
    green[5] = orange[5]
    green[6] = orange[6]
    # replace blue to orange
    orange[4] = blue[4]
    orange[5] = blue[5]
    orange[6] = blue[6]
    # replace red to blue
    blue[4] = red[4]
    blue[5] = red[5]
    blue[6] = red[6]
    # replace green to red
    red[4] = placeholder[0]
    red[5] = placeholder[1]
    red[6] = placeholder[2]
    #rotate yellow
    yellow = rotate(yellow)
    return white, red, green, blue, yellow, orange

def front(white, red, green, blue, yellow, orange):
    placeholder = [white[4], white[5], white[6]]
```

```
    # replace green to white
    white[4] = green[2]
    white[5] = green[3]
    white[6] = green[4]
    # replace yellow to green
    green[2] = yellow[0]
    green[3] = yellow[1]
    green[4] = yellow[2]
    # replace blue to yellow
    yellow[0] = blue[6]
    yellow[1] = blue[7]
    yellow[2] = blue[0]
    # replace white to blue
    blue[6] = placeholder[0]
    blue[7] = placeholder[1]
    blue[0] = placeholder[2]
    #rotate red
    red = rotate(red)
    return white, red, green, blue, yellow, orange

def move(cube, move_num):
    (white, red, green, blue, yellow, orange) = ungroup_cube(cube)
    if (move_num == 1):
        # print("up")
        (white, red, green, blue, yellow, orange) =
        up(white, red, green, blue, yellow, orange)
    elif (move_num == 2):
        # print("right")
        (white, red, green, blue, yellow, orange) =
        right(white, red, green, blue, yellow, orange)
    elif (move_num == 3):
        # print("left")
        (white, red, green, blue, yellow, orange) =
        left(white, red, green, blue, yellow, orange)
    elif (move_num == 4):
        # print("back")
        (white, red, green, blue, yellow, orange) =
        back(white, red, green, blue, yellow, orange)
    elif (move_num == 5):
        # print("down")
        (white, red, green, blue, yellow, orange) =
        down(white, red, green, blue, yellow, orange)
    elif (move_num == 6):
        # print("front")
        (white, red, green, blue, yellow, orange) =
        front(white, red, green, blue, yellow, orange)
```

```
    return object_cube(white, red, green, blue, yellow, orange)

def randomize_cube(cube):
    for i in range (0, 30):
        random_int = random.randint(1, 6)
        cube = move(cube, random_int)
    return cube

def object_cube(white, red, green, blue, yellow, orange):
    cube = [white, red, green, blue, yellow, orange]
    return cube

def ungroup_cube(cube):
    white = cube[0]
    red = cube[1]
    green = cube[2]
    blue = cube[3]
    yellow = cube[4]
    orange = cube[5]
    return white, red, green, blue, yellow, orange
```

6.2. **Code: Solver.py.**

```
import Cube

def add_to_color(char, r, g, b, w, o, y):
    if (char == 'W'):
        w += 1
    if (char == 'R'):
        r += 1
    if (char == 'G'):
        g += 1
    if (char == 'B'):
        b += 1
    if (char == 'Y'):
        y += 1
    if (char == 'O'):
        o += 1
    return r, g, b, w, o, y

def parity_cycle(cycle):
    par = 0
    passed = []
    for i in range(0, len(cycle)):
        first = cycle[i]
        if first not in passed:
            current = cycle[first]
```

```
            long = 0
            passed.append(first)
            while (first != current):
                passed.append(current)
                long += 1
                current = cycle[current]
            par += long

    # return 1 if even
    # return -1 if odd
    if (par % 2 == 0):
        return 1
    return -1

def fix_corner(corners):
    new = []
    for i in range(0, 8):
        if ('B' in corners[i] and 'Y' in corners[i] and
        'R' in corners[i]):
            new.append('BYR')
        elif ('B' in corners[i] and 'Y' in corners[i] and
        'O' in corners[i]):
            new.append('BYO')
        elif ('B' in corners[i] and 'W' in corners[i] and
        'O' in corners[i]):
            new.append('BWO')
        elif ('B' in corners[i] and 'W' in corners[i] and
        'R' in corners[i]):
            new.append('BWR')
        elif ('G' in corners[i] and 'W' in corners[i] and
        'R' in corners[i]):
            new.append('GWR')
        elif ('G' in corners[i] and 'Y' in corners[i] and
        'R' in corners[i]):
            new.append('GYR')
        elif ('G' in corners[i] and 'Y' in corners[i] and
        'O' in corners[i]):
            new.append('GYO')
        elif ('G' in corners[i] and 'W' in corners[i] and
        'O' in corners[i]):
            new.append('GWO')
    return new

def make_corner_arr(cube):
    (white, red, green, blue, yellow, orange) = Cube.ungroup_cube(cube)
    # 1 is byr
```

```
        # blue[6], yellow[2], red[4]
    # 2 is byo
        # blue[4], yellow[4], orange[6]
    # 3 is bwo
        # blue[2], white[2], orange[0]
    # 4 is bwr
        # blue[0], white[4], red[2]
    # 5 is gwr
        # green[2], white[6], red[0]
    # 6 is gyr
        # green[4], yellow[0], red[6]
    # 7 is gyo
        # green[6], yellow[6], orange[4]
    # 8 is gwo
        # green[0], white[0], orange[2]

    pos = [[6, 2, 4], [4, 4, 6], [2, 2, 0], [0, 4, 2], [2, 6, 0],
    [4, 0, 6], [6, 6, 4], [0, 0, 2]]
    corners = []

    for i in range(0, 4):
        string = blue[pos[i][0]]
        if (i == 0):
            string += yellow[pos[i][1]] + red[pos[i][2]]
        elif (i == 1):
            string += yellow[pos[i][1]] + orange[pos[i][2]]
        elif (i == 2):
            string += white[pos[i][1]] + orange[pos[i][2]]
        else:
            string += white[pos[i][1]] + red[pos[i][2]]
        corners.append(string)
    for i in range(4, 8):
        string = green[pos[i][0]]
        if (i == 4):
            string += white[pos[i][1]] + red[pos[i][2]]
        elif (i == 5):
            string += yellow[pos[i][1]] + red[pos[i][2]]
        elif (i == 6):
            string += yellow[pos[i][1]] + orange[pos[i][2]]
        else:
            string += white[pos[i][1]] + orange[pos[i][2]]
        corners.append(string)
    corners = fix_corner(corners)
    return corners

def cycle_corners(cube):
```

```
    corner_start = ['BYR', 'BYO', 'BWO', 'BWR', 'GWR', 'GYR', 'GYO', 'GWO']
    corners = make_corner_arr(cube)
    cycle = []
    for i in range(0, 8):
        cycle.append(corners.index(corner_start[i]))
    return cycle

def find_corner_par(cube):
    cycle = cycle_corners(cube)

    return parity_cycle(cycle)

def fix_edge(edges):
    new = []
    for i in range(0, 12):
        if ('B' in edges[i] and 'O' in edges[i]):
            new.append('BO')
        elif ('B' in edges[i] and 'Y' in edges[i]):
            new.append('BY')
        elif ('B' in edges[i] and 'W' in edges[i]):
            new.append('BW')
        elif ('B' in edges[i] and 'R' in edges[i]):
            new.append('BR')
        elif ('G' in edges[i] and 'W' in edges[i]):
            new.append('GW')
        elif ('G' in edges[i] and 'R' in edges[i]):
            new.append('GR')
        elif ('G' in edges[i] and 'Y' in edges[i]):
            new.append('GY')
        elif ('G' in edges[i] and 'O' in edges[i]):
            new.append('GO')
        elif ('O' in edges[i] and 'Y' in edges[i]):
            new.append('OY')
        elif ('O' in edges[i] and 'W' in edges[i]):
            new.append('OW')
        elif ('R' in edges[i] and 'Y' in edges[i]):
            new.append('RY')
        elif ('R' in edges[i] and 'W' in edges[i]):
            new.append('RW')
    return new

def make_edge_arr(cube):
    (white, red, green, blue, yellow, orange) = Cube.ungroup_cube(cube)
    # 1 is bo
        # blue[3], orange[7]
    # 2 is by
```

```
    # blue[5], yellow[3]
# 3 is br
    # blue[7], red[3]
# 4 is bw
    # blue[1], white[3]
# 5 is ow
    # orange[1], white[1]
# 6 is oy
    # orange[5], yellow[5]
# 7 is ry
    # red[5], yellow[1]
# 8 is rw
    # red[1], white[5]
# 9 is go
    # green[7], orange[3]
# 10 is gy
    # green[5], yellow[7]
# 11 is gr
    # green[3], red[7]
# 12 is gw
    # green[1], white[7]

pos = [[3, 7], [5, 3], [7, 3], [1, 3], [1, 1], [5, 5], [5, 1], [1, 5],
[7,3], [5, 7], [3, 7], [1, 7]]
edges = []

for i in range(0, 4):
    string = blue[pos[i][0]]
    if (i == 0):
        string += orange[pos[i][1]]
    elif (i == 1):
        string += yellow[pos[i][1]]
    elif (i == 2):
        string += red[pos[i][1]]
    else:
        string += white[pos[i][1]]
    edges.append(string)
for i in range(4, 6):
    string = orange[pos[i][0]]
    if (i == 4):
        string += white[pos[i][1]]
    else:
        string += yellow[pos[i][1]]
    edges.append(string)
for i in range(6, 8):
    string = red[pos[i][0]]
```

```
        if (i == 6):
            string += yellow[pos[i][1]]
        else:
            string += white[pos[i][1]]
        edges.append(string)
    for i in range(8, 12):
        string = green[pos[i][0]]
        if (i == 8):
            string += orange[pos[i][1]]
        elif (i == 9):
            string += yellow[pos[i][1]]
        elif (i == 10):
            string += red[pos[i][1]]
        else:
            string += white[pos[i][1]]
        edges.append(string)
    edges = fix_edge(edges)
    return edges

def cycle_edges(cube):
    edge_start = ['BO', 'BY', 'BR', 'BW', 'OW', 'OY', 'RY', 'RW', 'GO',
    'GY', 'GR', 'GW']
    edges = make_edge_arr(cube)
    cycle = []
    for i in range(0, 12):
        cycle.append(edges.index(edge_start[i]))
    return cycle

def find_edge_par(cube):
    cycle = cycle_edges(cube)

    return parity_cycle(cycle)

def orientation_num_edge(inface, adj):
    if (inface == 'W' or inface == 'Y'):
        return 0
    elif (adj == 'W' or adj == 'Y'):
        return 1
    elif (inface == 'R' or inface == 'O'):
        return 1
    elif (inface == 'G' or inface == 'B'):
        return 0

def sum_edges(cube):
    (white, red, green, blue, yellow, orange) = Cube.ungroup_cube(cube)
    sum = 0
```

```
    # verify white face
    for i in range(0, 4):
        adj = 0
        if (i == 0):
            adj = orange[1]
        elif (i == 1):
            adj = blue[1]
        elif (i == 2):
            adj = red[1]
        else:
            adj = green[1]
        sum += orientation_num_edge(white[2 * i + 1], adj)
    # verify yellow face
    for i in range(0, 4):
        adj = 0
        if (i == 0):
            adj = red[5]
        elif (i == 1):
            adj = blue[5]
        elif (i == 2):
            adj = orange[5]
        else:
            adj = green[5]
        sum += orientation_num_edge(yellow[2 * i + 1], adj)
    # verify remaining horizontal
    for i in range(0, 4):
        adj = 0
        face = 0
        if (i == 0):
            adj = red[7]
            face = green[3]
        elif (i == 1):
            face = green[7]
            adj = orange[3]
        elif (i == 2):
            face = blue[7]
            adj = red[3]
        else:
            face = blue[3]
            adj = orange[7]
        sum += orientation_num_edge(face, adj)
    if (sum % 2 != 0):
        return False
    return True

def orientation_num_corner(inface, clock, anti):
```

```
    if (inface == 'W' or inface == 'Y'):
        return 0
    elif (clock == 'W' or clock == 'Y'):
        return 2
    elif (anti == 'W' or anti == 'Y'):
        return 1

def sum_corners(cube):
    (white, red, green, blue, yellow, orange) = Cube.ungroup_cube(cube)
    sum = 0
    # verify white face
    for i in range(0, 4):
        clock = 0
        anti = 0
        if (i == 0):
            clock = green[0]
            anti = orange[2]
        elif (i == 1):
            clock = orange[0]
            anti = blue[2]
        elif (i == 2):
            clock = blue[0]
            anti = red[2]
        else:
            clock = red[0]
            anti = green[2]
        sum += orientation_num_corner(white[2 * i], clock, anti)
    # verify yellow face
    for i in range(0, 4):
        clock = 0
        anti = 0
        if (i == 0):
            clock = green[4]
            anti = red[6]
        elif (i == 1):
            clock = red[4]
            anti = blue[6]
        elif (i == 2):
            clock = blue[4]
            anti = orange[6]
        else:
            clock = orange[4]
            anti = green[6]
        sum += orientation_num_corner(yellow[2 * i], clock, anti)
    if (sum % 3 != 0):
        return False
```

```
    return True

def valid_cube(cube):
    # verify there are only eight cubies of each color
    (white, red, green, blue, yellow, orange) = Cube.ungroup_cube(cube)
    r = 0
    g = 0
    b = 0
    w = 0
    y = 0
    o = 0
    for i in range (0, 8):
        (r, g, b, w, o, y) = add_to_color(white[i], r, g, b, w, o, y)
        (r, g, b, w, o, y) = add_to_color(red[i], r, g, b, w, o, y)
        (r, g, b, w, o, y) = add_to_color(blue[i], r, g, b, w, o, y)
        (r, g, b, w, o, y) = add_to_color(green[i], r, g, b, w, o, y)
        (r, g, b, w, o, y) = add_to_color(orange[i], r, g, b, w, o, y)
        (r, g, b, w, o, y) = add_to_color(yellow[i], r, g, b, w, o, y)
    if (not (r == g == b == w == o == y == 8)):
        return False
    # if same sign
    if (find_corner_par(cube) != find_edge_par(cube)):
        return False
    # if number of twists is conserved
    if (not sum_corners(cube)):
        return False
    # if number of flips is conserved
    if (not sum_edges(cube)):
        return False
    return True

def solve(cube):
    if (not valid_cube(cube)):
        return False

    if (Cube.is_solved(cube)):
        print("The given cube is already solved!")
    while (not Cube.is_solved(cube)):
        move = 0
        # determine the next move
        # update move
        # cube = Cube.move(cube, move)
        return Cube.generate_cube()

    return cube
```

6.3. **Code: main.py.**

```
import Cube, Solver

def new_cube():
    cube = Cube.generate_cube()
    return cube

def randomize_cube(cube):
    return Cube.randomize_cube(cube)

def given_cube(cube):
    if (not Solver.valid_cube(cube)):
        print("The given cube is not valid!")
        Cube.print_cube(cube)
    else:
        cube = Solver.solve(cube)

def first_interaction():
    print("Welcome to Rubik's Cube Solver!")
    print("0: To close Rubik's Cube Solver")
    print("1: To generate a solved Rubik's Cube")
    print("2: To input a Rubik's Cube configuration")
    decision = input("Please enter your choice: ")

    if (decision == '0'):
        return -1
    if (decision == '1'):
        return new_cube()
    if (decision == "2"):
        give_success = terminal_give()
        if (give_success == -1):
            return 0
        else:
            return give_success

def valid_input(color):
    if (color == 'W' or color == 'R' or color == 'G' or color == 'B' or
    color == 'O' or color == 'Y'):
        return True
    return False

def ask_face():
    face = []
    for i in range(1, 9):
        while (True):
```

```
            input_color = input("%d cubie: " % (i))
            if (input_color == 'QUIT'):
                return -1
            if valid_input(input_color):
                break
            print("Give a valid value")
        face.append(input_color)
    return face

def terminal_give():
    print("Give the colors of the cubie as follows:")
    print("1. Red is the front, white up, and blue to the right.")
    print("2. The order is clockwise from the top left cubie")
    print("3. Do NOT include the center of the face,
    use those to orient the cube")
    print("4. White = W, Red = R, Green = G, Blue = B,
    Yellow = Y, Orange = O")
    print("To quit, type 'QUIT'")
    # ask for white face
    print("\tWhite Face:")
    white = ask_face()
    if (white == -1):
        return -1
    # ask for red face
    print("\tRed Face:")
    red = ask_face()
    if (red == -1):
        return -1
    # ask for green face
    print("\tGreen Face:")
    green = ask_face()
    if (green == -1):
        return -1
    # ask for orange face
    print("\tOrange Face:")
    orange = ask_face()
    if (orange == -1):
        return -1
    # ask for blue face
    print("\tBlue Face:")
    blue = ask_face()
    if (blue == -1):
        return -1
    # ask for Yellow face
    print("\tYellow Face:")
    yellow = ask_face()
```

```
    if (yellow == -1):
        return -1
    cube = Cube.object_cube(white, red, green, blue, yellow, orange)
    if (Solver.valid_cube(cube)):
        return cube
    else:
        print("The given cube was invalid")
        Cube.print_cube(cube)
        return -1

def terminal_has_cube(cube):
    while (True):
        print("0: Destroy the cube")
        print("1: Solve the cube")
        print("2: Make a move")
        print("3: Randomize the cube")
        choice = input("Please enter your choice: ")
        if (choice == '0'):
            print("Cube was deleted!")
            Cube.print_cube(cube)
            return -1
        if (choice == '1'):
            cube = Solver.solve(cube)
            Cube.print_cube(cube)
        if (choice == '2'):
            cube = make_move(cube)
        if (choice == '3'):
            cube = randomize_cube(cube)
            Cube.print_cube(cube)

def make_move(cube):
    while (True):
        print("0: To return to menu for the cube")
        print("1: Up/white rotation")
        print("2: Right/blue rotation")
        print("3: Left/green rotation")
        print("4: Back/orange rotation")
        print("5: Down/yellow rotation")
        print("6: Front/red rotation")
        choice = input("Please enter your choice: ")
        if (choice == '0'):
            return cube
        if (choice == '1'):
            print("Moved the White face")
            cube = Cube.move(cube, 1)
            Cube.print_cube(cube)
```

```
        elif (choice == '2'):
            print("Moved the Blue Face")
            cube = Cube.move(cube, 2)
            Cube.print_cube(cube)
        elif (choice == '3'):
            print("Moved the Green Face")
            cube = Cube.move(cube, 3)
            Cube.print_cube(cube)
        elif (choice == '4'):
            print("Moved the Orange Face")
            cube = Cube.move(cube, 4)
            Cube.print_cube(cube)
        elif (choice == '5'):
            print("Moved the Yellow Face")
            cube = Cube.move(cube, 5)
            Cube.print_cube(cube)
        elif (choice == '6'):
            print("Moved the Red Face")
            cube = Cube.move(cube, 6)
            Cube.print_cube(cube)

while(True):
    choice = first_interaction()
    if (choice == -1):
        print("Goodbye!")
        break
    elif (choice != 0):
        Cube.print_cube(choice)
        terminal_has_cube(choice)
```

REFERENCES

[1] JJ Chen. Group theory and the rubik's cube, 2004.
[2] Keith Conrad. The sign of a permutation.
[3] Courtney Cooke. Solving the rubik's cube using group theory. 2017.
[4] Lindsey Daniels. Group theory and the rubik's cube. *Lakehead Universityc*, 2014.
[5] Sean Harrell. Rubik's cubes and group theory. 2012.

PURDUE UNIVERSITY, WEST LAFAYETTE, IN, USA
Email address: mdsantia@purdue.edu