EXPLORATION OF THE POCKET RUBIK'S GROUP

RYAN BROWN

Abstract. The Pocket Rubik's group has more than 88 million elements. The inherit symmetry of this group, can cut down the number of computations needed to be done for full analysis of the group. The comparatively small number of equivalence classes then can be analyzed to fully comprehend the behavior of the group. This paper will utilize the programming language Julia to partially map out the Cayley graph.

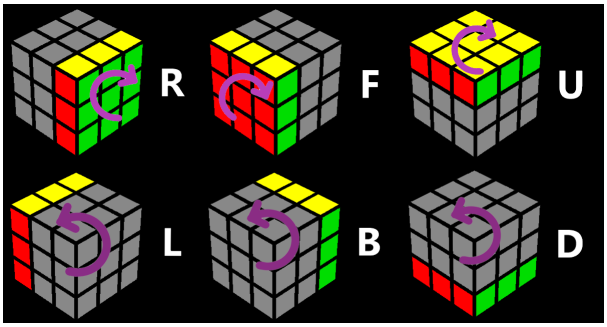
1. Introduction

The Rubik's group is defined as any possible permutation of the cube and as such has a cardinality of 4.352×10^{19}. Each element of the group is one possible position of the cube. The operations on the group are simply the different turns one can perform to the cube. Using this definition, the mathematical exploration of the group becomes the next step.

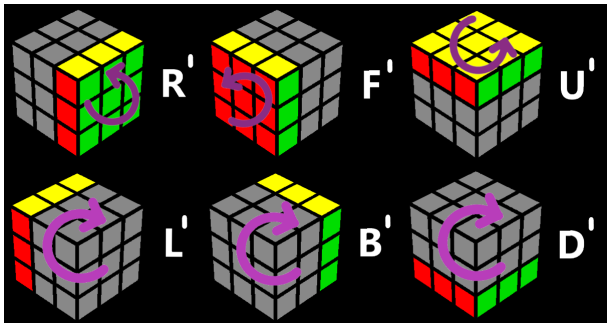
A pocket cube is the $2 \times 2 \times 2$ version of the Rubik's cube. This paper will then define the pocket Rubik group as the group of permutations of the pocket cube. The 8 individual cubes, cubies[4], can be shuffled in 8! ways, then fix one cube in space and the other 7 cubies can be rotated in 3 ways. Meaning there are $8! \times 3^7 = 88,179,840$ elements.

Date: April 2022.

There are 12 operations or turns which can be performed on the cube. Each turn is defined by the face which is rotated, based on the view of the solver of the cube. The moves R, L, F, B, U, and D are rotating the right, left, front, back, upper, bottom, and down face clockwise $\frac{\pi}{4}$ or a quarter turn. The inverse moves, R', L', etc. rotates the face corresponding to the letter counter clockwise a quarter turn. These turns are displayed below.



[1]



[1]

Finally 2 quarter turns can be written as 2F, 2L, etc. A sequence of moves then can be defined as a string of these letters and numbers. The method for labeling turns is called Singmaster Notation[4].

One of the debates inside of the study of the Rubik group is whether to count a half-turn as one turn or two. This can be described as the quarter-turn metric or the half-turn metric. East Coast researchers

often chose quarter-turn metric, while the West Coast chose half-turn [6]. This paper will utilize the quarter-turn metric.

If one were to give God a Rubik's cube, they would solve it in the minimal moves necessary. God's number is the maximum number of moves that god would need to use to solve any given position of the Rubik's cube. In group theory this can be referred to as the diameter of the Rubik's Group. Much research and computation time has been put into calculating God's number for the Rubik's cube. Using the half-turn metric, God's number is 20 [6], where the quarter-turn metric has a God's number of 26 [5].

The pocket cube, however, has significantly less research done on it. From a old mailing list, it is claimed that God's Number for the half-turn metric pocket cube is 11, where the quarter-turn metric yields 14.[2] While these numbers seem plausible, I embrace a good amount of scepticism due to the lack of work as well as the location of the information.

This paper explores the roughly 88 million elements of the pocket Rubik group in the hopes of answering the question, 'When is a pocket cube properly shuffled?' To find the answer to this question, first it is prudent to reduce the number of elements that need to be examined by utilizing the inherent symmetry inside of the pocket Rubik group. Using analysis, we can answer several questions relating to our main questions, however computation will be utilized to generate a Markov Chain of elements. This Markov chain will then be used to analyze how quickly the probable cube positions disperse.

Given the limited time of this project as well as the multitude of commitments I have during the semester, this paper concludes prematurely with a limited Cayley Graph. The possible next steps in this project will be described at the end of this paper and will be used as a guide for some work through the summer and possibly beyond. These questions may also serve as a guide to other students or people interested in researching the Pocket Cube or similar concepts.

2. Pocket Rubik's Group Symmetry

Elements inside the Pocket Rubik's group are called symmetric if they can be solved using equivalent methods. A group of all elements that are symmetric is called an equivalence class. A equivalence class can be represented by it's canonical representative, which is chosen by a rule or algorithm prescribed by the researcher. When studying this group, it will be sufficient to study the subgroup with elements that are the canonical representatives.

There are four types of symmetries of the pocket Rubik group, cube isometries, color permutations, cube reflections, and color reflections. However, cube reflection and color reflections must occur with one another, so including both is mute.

Cube isometries have a 24-fold symmetry on the Pocket Rubik's Group. The idea behind the symmetry is a cube rotated in space must be the same exact cube by any reasonable metric. Each face can be oriented in 1 of 6 positions, with any one adjacent face being oriented in 1 of 4, which creates the 24-fold symmetry.

Color permutations have 24-fold symmetry. If two cubes can be solved by using the same moves, then clearly, they are practically the same cube. Color permutations helps remove cubes which are practically the same for all purposes except for the arrangement of the colors. One color (say yellow) can go to any other color. The opposite color of the first (white) must then go to the opposite color of the first. This leaves four open colors. A second color (say blue) can go to any of the remaining four colors, while the opposite of this second color (green) must go to the opposite. Finally the remaining two colors, must then be placed in the arrangement to ensure the cube is properly colored. This creates the 24-fold symmetry.

The 2-fold Cube/color reflections simply is what happens if the cube is reflected through a mirror. The cube is practically the same yet different.

The code utilizes both cube isometries and color permutations to reduce the number of computations needed. Every cube isometry and color permutation are not unique however. For example the identity (solved) state of the cube does have 24 isometries. However the 24 color permutations each simply map to one of the isometries. This overlapping means we do not have the full 576-fold symmetry entailed.

We will call the i-equivalence classes, equivalence classes utilizing only isometries of the cube. Each equivalence class will have exactly 24 elements (see proof in section 3), and there will be $\frac{88,179,840}{24} = 3,674,160$ equivalence classes.

We will call the ip-equivalence classes, equivalence classes utilizing both isometries of the cube as well as color permutations. As isometries cannot overlap (see proof in section 3), the only combinations will be combining isometries which are equivalent over color permutations. As stated above the solved state can be represented by only 1 i-equivalence class, and the ip-equivalence class is identical as the color permutations overlap with the isometries. One move out from solved is represented by 6 i-equivalence classes while only 2 ip-equivalence classes are needed.

3. Essential Proofs

Lemma 1. *Turns are equivalent regardless of isometry.*

From any position there are 12 turns that can be made. These moves rotate only half the cube about one of the axes either clockwise or counterclockwise. There are 3 axes, so 6 turns in one of 2 directions which gives our 12 moves. Each isometry can be thought of as rotating the entire cube. When you rotate the entire cube, turns simply change definition. For example a F could become a R under a different Isometry. Each turn can only move 4 cubies, and there are 12 distinct ways to move 4 cubies using a pocket cube. The turns therefore are just relabeled for every Isometry. ■

Let F*, F*′, etc. be defined such that on a canonical representative the move is, F, F', etc. and any element after having F*, F*′, etc. performed on it will move to the same equivalence class as the canonical representative does with F, F', etc. That way one can talk about a move consistently between equivalence classes.

Lemma 2. *Performing a turn on a cube changes the i-equivalence class*

I-equivalence classes utilize only isometries of the cube. For a turn to occur 4 cubies must move, but isometries changes the location of all 8 cubies. As such you cannot possibly apply a turn, change isometries and stay at the same i-equivalence as you cannot perform a isometry and get back the same element as you put in. ■

Lemma 3. *Each i-equivalence class has 24 element*

Using similar logic one isometry moves all 8 cubies. Every position has some arrangement of 8 distinct cubies. Earlier it is described that we orientate two faces, which gives us the 24-fold symmetry for each cube. Orientating faces is equivalent to orientating cubies. Orientating two cubies fixes the position of the other 6 as they must move as a group. As they are all distinct cubies, the 24 places to put two cubies must therefore generate 24 unique elements and only those elements. ■

Lemma 4. *Right or right inverse turn on a random element of an i-equivalence class is equivalent to a random move on the canonical representative*

Now suppose you toss a cube in the air, and it lands uniformly randomly on some element of the i-equivalence class. From Lemma 3 all i-equivalence classes contain exactly 24 elements. You then perform a right clockwise turn. Perhaps D. In the conical representative this is some face rotated clockwise. Perhaps F^*. Performing D to the random element in the i-equivalence class will then yield a element in the

same class as performing F^* on the canonical representative of the first random element. Isometries don't change the i-equivalence classeses between turns, just the element in those classes as they are not necessarily the same.

However via isometries of the cube it is possible to perform a turn, and a isometry such that it would be equivalent to performing a 'partner turn' and a different isometry to get the same cube. For example performing R and performing L along with some isometries will yield the same cube. Partner turns are turns that operate along the same axis, but opposite sides of the cube (R-L, D-U, F-B). So $\frac{2}{12} = \frac{1}{6}$ of the clockwise turns go to the same equivalence class. Lemma 1 and lemma 2 state the turns are the same regardless of isometries and that the turns of the isometries go to the same i-equivalence classes and we just showed the probabilities are the same of a random clockwise turn and R on a random element in hte i-equivalence class. As such randomly choosing at random a face and rotating clockwise (some random clockwise turn) is equivalent to picking a random equivalence class and performing a right move. It follows that if you randomly perform a clockwise or counterclockwise right turn to a random element in the equivalence class, then that is the same as performing a random move on the canonical representative. ■

This enables us to explore the elements via a Markov chain while only looking at the canonical representative as we are now certain it exhibits the behaviors of all the elements of the i-equivalence class.

Corollary 4.1. *I-equivalence classes must move between each other with some even number of edges connecting them in the Cayley Graph.*

This is due to the 'partner' turns described earlier. Performing one move and a isometry is equivalent to the other and a different isometry. Even numbers simply are multiples of twos. The edges of the Cayley Graph simply refer to the number of turns between equivalence classes.

Corollary 4.2. *A i-equivalence class can only have edges to a maximum of 6 other equivalence classes on the Cayley Graph.*

Every turn has its partner turn. Therefore the 12 turns are reduced down to 6 that could possibly go to different equivalence classes.

Lemma 5. *IP-equivalence classes must be a combination of i-equivalence classes and therefore have some multiple of 24 elements.*

If two elements are equivalent via color permutations, then it follows that their i-equivalence class canonical representatives must be equivalent via color rotation and isometry. Therefore the two classes must be equivalent in ip-equivalence. As all i-equivalence classes have 24 elements, ip-equivalence classes must have some multiple of 24 elements.
■

Corollary 5.1. *IP-equivalence classes must be a combination of i-equivalence classes at the same depth from solved on the Cayley Graph*

If two i-equivalence classes are equivalent via color permutation, then you must be able to solve them via equivalent moves and therefore the depth must be equivalent.

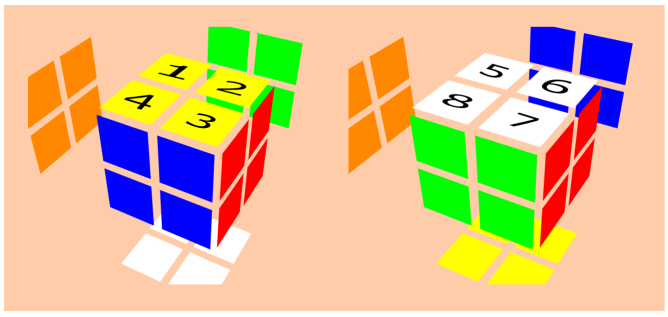
Remark. *On the Cayley graph edges can only move towards solved (lower depth) by one, away from solved (higher depth) by one, or at the same level(same depth).*

This is apparent once some examination of the graph is done. if a edge connects to a depth more than one above, then the depth of that element is wrong as it must be closer than originally thought. If a edge connects to a depth more than one below, then the edge below must be at depth one below. There is nothing wrong with a edge between cubes at the same level inherently, as it would neither increase nor decrease the minimum distance to solved.

4. Code Explanation

While the code is present in the appendix (7.3), several important functions will be described below.

The base code was created by Guido Dipierto in 2020 [3]. A cube is defined as two 8×1 arrays. The first array is some permutation of [1,2,3,4,5,6,7,8] and gives the location of each cubie as defined by the picture below. This is described as Speffz Shceeme of labeling the cube.



[3]

The second array states the orientation of each cubie and whether it is not rotated (0), rotated clockwise (1), or rotated counterclockwise (2).

Dipierto then defines the 6 clockwise quarter moves one can perform on a cube, and the inverses can be generated via a composition of a turn three times. He also defined X, Y, and Z full cube Isometry.

The first function I created is entitled print_cube and it prints out the cube. It translate each position as to where each cubie is, and gets the color at any particular point by referencing the color table (shown below).

orr	1	2	3	4	5	6	7	8
0	*y*	*y*	*y*	*y*	*w*	*w*	*w*	*w*
1	*g*	*r*	*b*	*o*	*b*	*r*	*g*	*o*
2	*o*	*g*	*r*	*b*	*o*	*b*	*r*	*g*

It then prints the cube in the form below. In this example the green face at the bottom represents the back shifted down, while the white represents the down face, orange left, and so on.

```
    \  y y  /
     \ y y /
  o o | b b | r r
  o o | b b | r r
     / w w \
    /  w w  \


      g g
```

```
g g
```

The Orientate cube function merely rotates the cube until the BYO cubie is in position 4 with yellow on top (orientation 0).

The next several functions deal with assigning a number to each of the cube elements. The functions both can generate a cube's number given the cube and generate the cube given the number in constant time.

The two aspects of the cube are the position array and the orientation array. The orientation array can be described as a base three number (or trinary number) and as such has 3^8 different possibilities. The positional array permutes 8 unique elements, as such there as 8! possibilities. In total there are $8! \times 3^8$ possibilities of the cube structures or $264,539,520$ permutations. Only $\frac{1}{3}$ of those elements are actual elements of the cube and $\frac{1}{72}$ are canonical representatives, however by utilizing a sparse matrix to store entries, the excess elements should provide minimal losses to computer memory.

The orientation matrix is transformed into a number by looking at it as a trinary number. The 8! Permutations are generated in lexicographical by the computer. By reverse engineering the process a number can be generated given a permutation of the array. This is accomplished by subtracting the minimum of the array from the first number in the array multiplying the difference by the factorial of the size of the array minus one, then adding that to the position number (starts at zero), decrementing the entire array by the first element (i.e. moving high numbers than the first element down one), then recursing. This always

runs 8 recursions and is therefore constant time. This allows us to find the lexographical number of the position array.

The cube's position is then calculated by finding the position number, multiplying that by 3^8 and adding the orientation number

In order to find the canonical representative of the ip-equivalence class we orientate all the color permutations, then take the minimum number generated in the method above.

A cube class was created which had each of the 24 color permutations of the identity to perform moves on. This was simpler than creating a formula which swapped colors as the storage structure is not concussive a formula. The reason being is that one would have to add or subtract extra from cubes based on where the yellow/white face ended up, which is based on the moves performed. The class containing the permutations then could perform any sequence of moves on all the cubes, then give the canonical representative number for the i-equivalence classes and the lowest becomes the canonical Representative for the ip-equivalence class.

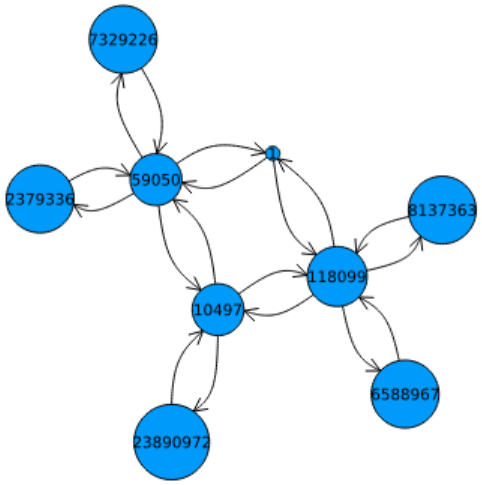
The julia programming language performs the turn functions quickly, as such I determined that instead of storing all the cubes, the turn sequence would be stored and added to, to save on memory (as we would be visiting a lot of nodes). The current version still does not adequately utilize the symmetry of the cube to reduce computation power, but it is close.

Using a online service, the program can generate roughly 100 canonical representatives before running out of memory. The average element

in the Cayley Graph then takes 1.3 seconds of computation time as it accomplished this in 2 hours and 10 minutes. The full graph would then likely take somewhere between 2.2 days and 5.3 days depending on how overlapped the isometries and permutations are. This memory problem occurred twice, once with utilizing a sparse matrix, and again when simply writing the elements to a log. The memory error occured both times at 2 hours and 10 minutes, which is interesting. Regardless with some more optimization and prehaps a better place to run the code, it will generate the full cayley graph.

5. Partial Cayley Graph and Explanation

A Cayley graph going down 2 in depth is pictured below.



The Cayley graph shows exactly what is expected. The numbers represent the numbers generated by the method described in the previous section +1 (so we start at one, rather than zero). Utilizing ip-equivalence classes, we can see that from the identity there are two

elements, either a clockwise turn or a counter-clockwise turn as via color rotations they are all identical. The half turn generates from either one of those two elements, and the two elements also each have two unique elements they spread from as well.

For the sake of space the 100 elements calculated will be excluded from this paper, as it will simply be ugly to look at as well as next to impossible to judge from. Even generating a graph will look decently ugly. As no further work is done beyond a larger version of the Cayley Graph, additional work will not be shown.

6. Further Questions and Direction

I hope to continue this research into the summer and perhaps beyond. This research over the semester simply did not provide me with ample time to study the cube and answer my multitude of questions. This further exploration into group theory via computation will enable me to both experience further math research as well as produce results that can be used by Rubik Cube Enthusiasts. More generally however, this process can be applied to other groups, and study how groups disperse with given operations.

Several clear options exist for the immediate next step.

The first is to develop a method to increase the symmetry utilized by creating a program which can generate reflected cubes. This will only decrease the possible computation size by half, but half is still significant.

A second is to better optimize my code which will allow a complete Cayley graph to be found. This full graph will either confirm or deny

God's number on the pocket cube, as well as give a basis for analysis of the group.

My initial question asks how quickly does the group disperse, this can be accomplished by studying a generated matrix with a markov chain to see how quick the probabilities distribute throughout the graph. This can be studied by taking the difference between any two steps on the markov chain (via some induced norm) and seeing when the differences start to become minimal. It will be important to think about which norm to use, as well as what each will actually tell us.

Continuing along this theme, this tells us how much we should shuffle for God, and for now I am not handing a pocket cube to God. So to better achieve a metric for when a shuffle is good enough, we should assign weights to each node, based on how quick any given algorithm can solve that equivalence class. This dispersion may result in a similar answer, but one is unsure until it is shown.

One could also shuffle a different way, instead of randomly picking 1 of 12 moves, perhaps it is only 1 of 4, or perhaps some set sequence which is repeated over and over again, or a small set of random sequences. Each of these will generate completely different steps along the Markov chain and may give us a quicker or similar method of shuffling the cube. In the very least humans aren't likely to pick a random number between 1 and 12, but between 1 and 3 or 1 and 2 is much more doable (though not completely possible for a human, much more feasible).

There are also more directions to take this project. For example to compete with God's number, the devil has their own. The Devil's Algorithm is a sequence of moves which must go through every element in the group as a circuit, thus you can use this algorithm regardless of starting position and eventually get to solved. The Devil's number is the number of moves in the shortest algorithm. This number lies between 100 thousand and 2.19 million. [7]

Another idea would be to generate a problem which performs a walk through a group in real time, which would be able to be generalized to any group any particular person may be working on. While this may not be the best idea for this project given the size of the group, this real time walk may have many uses for smaller groups where people may want a Markov chain fast (Probability application may already have a similar thing, but would still be neat).

In conclusion I hope and plan to continue this work. I am looking forward to sharing any and all conclusions I find.

7. Appendix

7.1. **Code.** Code here is the code I used in case others would like to examine it for inaccuracies, run time improvements, or general curiosity. The ∘ symbol is replaced with o below, I apologize for any confusion.

```
#########################
# Adding Packages #
#########################
import Pkg
```

```
Pkg.add("Combinatorics")
Pkg.add("BenchmarkTools")
Pkg.add("LoopVectorization")
Pkg.add("LinearAlgebra")
#########################
# Includes #
#########################
using Combinatorics
using BenchmarkTools
using LoopVectorization
using LinearAlgebra
using Base.Threads: @threads, @sync, @spawn
#########################
# Guido Dipietro - 2020 #
#########################
cp = (1,2,3,4,5,6,7,8) # like,speffz
co = (0,0,0,0,0,0,0,0) # 1 = clockwise
cube = (cp, co)
# Atoms
const Rp((a,b,c,d,e,f,g,h)) = (a,c,f,d,e,g,b,h)
const Ro(a) = @. (a + (0,1,2,0,0,1,2,0)) % 3
const Up((a,b,c,d,e,f,g,h)) = (d,a,b,c,e,f,g,h)
const xp((a,b,c,d,e,f,g,h)) = (d,c,f,e,h,g,b,a)
const xo(a) = @. (a + (2,1,2,1,2,1,2,1)) % 3
# Movedefs
```

```
const R((p,o)) = Rp(p), (RooRp)(o)

const U((p,o)) = Up(p), Up(o)

const x((p,o)) = xp(p), (xooxp)(o)

const D = x o x o U o x o x

const y = U o D o D o D   ## Test if composition is saved, or matrix algebra do

const F = x o x o x o U o x

const B = y o y o F o y o y

const z = F o B o B o B

const L = y o y o R o y o y

#########################

# Ryan Brown- 2022 #

#########################

mutable struct DIRECTED_GRAPH

    from::Int

    to::Vector{Int}

    times::Vector{Int8}

end

mutable struct CUBE

    c1::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}

    c2::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}

    c3::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}

    c4::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}

    c5::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}

    c6::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}

    c7::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
```

```
    c8::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c9::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c10::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c11::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c12::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c13::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c14::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c15::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c16::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c17::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c18::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c19::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c20::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c21::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c22::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c23::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
    c24::Tuple{NTuple{8, Int64}, NTuple{8, Int64}}
end
function INIT_CUBE()
    return CUBE(cube,(transforms[1],(0,0,0,0,0,0,0,0)),(transforms[2],(0,0,0,0,
end
function turns(cube::CUBE,move)
    c = INIT_CUBE()
    c.c1 = (move)(cube.c1)
    c.c2 = (move)(cube.c2)
```

```
    c.c3 = (move)(cube.c3)
    c.c4 = (move)(cube.c4)
    c.c5 = (move)(cube.c5)
    c.c6 = (move)(cube.c6)
    c.c7 = (move)(cube.c7)
    c.c8 = (move)(cube.c8)
    c.c9 = (move)(cube.c9)
    c.c10 = (move)(cube.c10)
    c.c11 = (move)(cube.c11)
    c.c12 = (move)(cube.c12)
    c.c13 = (move)(cube.c13)
    c.c14 = (move)(cube.c14)
    c.c15 = (move)(cube.c15)
    c.c16 = (move)(cube.c16)
    c.c17 = (move)(cube.c17)
    c.c18 = (move)(cube.c18)
    c.c19 = (move)(cube.c19)
    c.c20 = (move)(cube.c20)
    c.c21 = (move)(cube.c21)
    c.c22 = (move)(cube.c22)
    c.c23 = (move)(cube.c23)
    c.c24 = (move)(cube.c24)
    return c
end
function orientate_CUBE(cube::CUBE)
```

```
    cube.c1 = orientate_cube(cube.c1)
    cube.c2 = orientate_cube(cube.c2)
    cube.c3 = orientate_cube(cube.c3)
    cube.c4 = orientate_cube(cube.c4)
    cube.c5 = orientate_cube(cube.c5)
    cube.c6 = orientate_cube(cube.c6)
    cube.c7 = orientate_cube(cube.c7)
    cube.c8 = orientate_cube(cube.c8)
    cube.c9 = orientate_cube(cube.c9)
    cube.c10 = orientate_cube(cube.c10)
    cube.c11 = orientate_cube(cube.c11)
    cube.c12 = orientate_cube(cube.c12)
    cube.c13 = orientate_cube(cube.c13)
    cube.c14 = orientate_cube(cube.c14)
    cube.c15 = orientate_cube(cube.c15)
    cube.c16 = orientate_cube(cube.c16)
    cube.c17 = orientate_cube(cube.c17)
    cube.c18 = orientate_cube(cube.c18)
    cube.c19 = orientate_cube(cube.c19)
    cube.c20 = orientate_cube(cube.c20)
    cube.c21 = orientate_cube(cube.c21)
    cube.c22 = orientate_cube(cube.c22)
    cube.c23 = orientate_cube(cube.c23)
    cube.c24 = orientate_cube(cube.c24)
    return cube
```

```
end

#Constants

color_table = [['y','g','o'],['y','r','g'],['y','b','r'],['y','o','b'],['w','b'

const transforms = [

    ##Not Shown Y->Y, B->B

    (4,1,2,3,6,7,8,5), ##Y->Y B->O

    (3,4,1,2,7,8,5,6), ## B->G

    (2,3,4,1,8,5,6,7), ## B->R

    (6,5,4,3,2,1,8,7), ## Y->B, B->Y

    (5,8,1,4,3,2,7,6), ##B->O

    (8,7,2,1,4,3,6,5), ##B->W

    (7,6,3,2,1,4,5,8), ## B->R

    (1,8,7,2,3,6,5,4), ## Y->O, B->B

    (2,7,6,3,4,5,8,1), ## B->Y

    (3,6,5,4,1,8,7,2), ## B->G

    (4,5,8,1,2,7,6,3), ## B->W

    (2,1,8,7,6,5,4,3), ## Y->G, B->W

    (3,2,7,6,5,8,1,4), ##B->O

    (4,3,6,5,8,7,2,1), ## B->Y

    (1,4,5,8,7,6,3,2), ## B-> R

    (8,1,4,5,6,3,2,7), ## Y->R,B->B

    (7,2,1,8,5,4,3,6), ## B -> Y

    (5,4,3,6,7,2,1,8), ## B -> G

    (6,3,2,7,8,1,4,5),  ## B -> W

    (7,8,5,6,3,4,1,2), ## Y->w, B->B
```

```
    (6,7,8,5,4,1,2,3), ## B->O
    (5,6,7,8,1,2,3,4), ## B ->g
    (8,5,6,7,2,3,4,1)] ## B ->r
const arr_moves = [R,RoRoR,L,LoLoL,U,UoUoU,D,DoDoD,F,FoFoF,B,BoBoB]
const Arr1 = collect(permutations((1,2,3,4,5,6,7,8)));
##Position is:
#0-Top/Bottom
#1-Right of Top
#2-Left of Top
function color(position, cubie,orientation)
   return color_table[cubie][1+mod(orientation+position,3)]
end
function print_cube(cube)
    print_cube(cube[1],cube[2])
end
function print_cube(cp,co)
    print("   \\  ",color(0,cp[1],co[1]),' ',color(0,cp[2],co[2]))
    print("  /","",'\n',"    \\ ",color(0,cp[4],co[4]),' ',color(0,cp[3],co[3])
    print(color(2,cp[1],co[1])," ",color(1,cp[4],co[4])," | ",color(2,cp[4],co[
    print(color(1,cp[8],co[8])," ",color(2,cp[5],co[5])," | ",color(1,cp[5],co[
    print("    / ",color(0,cp[5],co[5])," ",color(0,cp[6],co[6])," \\",'\n')
    print("   /  ",color(0,cp[8],co[8])," ",color(0,cp[7],co[7]),"  \\ ",'\n','
    print("      ",color(1,cp[1],co[1])," ",color(2,cp[2],co[2]),'\n')
    print("      ",color(2,cp[8],co[8])," ",color(1,cp[7],co[7]))
end
```

```
# Orientate cube by cubie 4 in position 4, yellow ontop
function orientate_cube(cube)
    if(cube[1][4]==4)
    elseif(cube[1][3] == 4 || cube[1][6]==4 || cube[1][5]==4)
        while(cube[1][4]!=4)
            cube = z(cube)
        end
    elseif(cube[1][1]==4 || cube[1][8]==4)
        while(cube[1][4]!=4)
           cube = x(cube)
        end
    else
        cube = y(cube)
        return orientate_cube(cube)
    end
    if(cube[2][4] == 0)
        return cube
    elseif(cube[2][4]==1)
        cube = (yoyoyox)(cube) #y inverse
    else
        cube = (xoxoxoy)(cube) #x inverse
    end
end
##CANONICAL REPRESENTATIVES
function canonical_representative(cube::CUBE)
```

```
class = Vector(undef,24)
cube_n = orientate_CUBE(cube)
class[1] = cube_to_pos(cube_n.c1)+1
class[2] = cube_to_pos(cube_n.c2)+1
class[3] = cube_to_pos(cube_n.c3)+1
class[4] = cube_to_pos(cube_n.c4)+1
class[5] = cube_to_pos(cube_n.c5)+1
class[6] = cube_to_pos(cube_n.c6)+1
class[7] = cube_to_pos(cube_n.c7)+1
class[8] = cube_to_pos(cube_n.c8)+1
class[9] = cube_to_pos(cube_n.c9)+1
class[10] = cube_to_pos(cube_n.c10)+1
class[11] = cube_to_pos(cube_n.c11)+1
class[12] = cube_to_pos(cube_n.c12)+1
class[13] = cube_to_pos(cube_n.c13)+1
class[14] = cube_to_pos(cube_n.c14)+1
class[15] = cube_to_pos(cube_n.c15)+1
class[16] = cube_to_pos(cube_n.c16)+1
class[17] = cube_to_pos(cube_n.c17)+1
class[18] = cube_to_pos(cube_n.c18)+1
class[19] = cube_to_pos(cube_n.c19)+1
class[20] = cube_to_pos(cube_n.c20)+1
class[21] = cube_to_pos(cube_n.c21)+1
class[22] = cube_to_pos(cube_n.c22)+1
class[23] = cube_to_pos(cube_n.c23)+1
```

```
    class[24] = cube_to_pos(cube_n.c24)+1
    return minimum(class)
end
function number_to_trinary_array(num)
    array = [0,0,0,0,0,0,0,0]
    for i in 1:8
        array[i]=num%3
        num = floor(num/3)
    end
    return Tuple(array)
end
function trinary_array_to_number(array)
    num =0;
    for i in 1:8
        num += array[i]*3^(i-1)
    end
    return num
end
function pos_to_cube(pos)
   or_arr = number_to_trinary_array(pos%(3^8))
   pos_arr = Tuple(Arr1[convert(Int,floor(pos/(3^8))+1)])
   return (pos_arr,or_arr)
end
function decrement_array(array,num)
    array=collect(array)
```

```
    for i in 1:(size(array)[1])
        if(array[i]>num)
            array[i] = array[i]-1
        end
    end
    return Tuple(array)
end
function array_to_num(array)
    pos= (array[1]-findmin(array)[1])*factorial(7)
    for i in 1:6
        array = decrement_array(array[2:(9-i)],array[1])
        pos += (array[1]-findmin(array)[1])*factorial(7-i)
    end
    return pos
end
function cube_to_pos(cube)
   return array_to_num(cube[1])*3^8+trinary_array_to_number(cube[2])
end
function moves(cube)
    array_pos = Vector{Int}(undef, 12)
    @threads for i in 1:12
        array_pos[i] = cube_to_pos((arr_moves[i])(cube))+1
    end
    return array_pos
end
```

```
function add_moves(v)
    new_v = Vector(undef,12)
    @threads for i in 1:12
        new_v[i] = voarr_moves[i]
    end
    return new_v
end
function add_to_directed_graph(cube_pos, array_pos, dir)
    d_n = DIRECTED_GRAPH(cube_pos,unique(array_pos),[count(==(element),array_po
    push!(dir,d_n)
    return dir
end
## Recursively Adds, Took too much time, uses a lot of memory
#function recursive_add(cube,directed_graph,depth,deep)
#    cube_pos = cube_to_pos(cube);
#    if(!isempty(nonzeros(directed_graph[cube_pos+1,:])))
#        return
#    end
#    if(depth >= deep) return end
#   for i in 1:12
#        new_cube = arr_moves[i](cube)
#        directed_graph[cube_pos+1,cube_to_pos(new_cube)+1]=1
#        recursive_add(new_cube,directed_graph,depth+1,deep)
#    end
#end
```

```
#
#function generate_recursive(deep)
#    directed_graph = spzeros(264539520,264539520)
#    recursive_add(pos_to_cube(0),directed_graph,0,deep)
#    return directed_graph
#end
function generate_itterative(depth)
    been_to = BitVector(undef,264539520)
    cube = INIT_CUBE()
    been_to[1] = 1
    output = arr_moves
    output_cubes = Vector{Int}(undef,length(output));
    @threads for i in 1:12
        output_cubes[i] = canonical_representative(turns(cube,(output[i])))
    end
    num =0
    @info "NEW ROW"  num (1,unique(output_cubes),[count(==(element),output_cube
    num +=1
    for  i in  1:depth
        empty = 0
        output_new = Array{Any}(undef,length(output),12)
        output_new_cubes =  fill(-1,(length(output),12))
        @threads for j in 1:length(output)
            if(output_cubes[j] != -1)
                if(been_to[output_cubes[j]]==0)
```

```
                been_to[output_cubes[j]] = 1
                output_new[j,:] = add_moves(output[j])
                for move in 1:12
                    output_new_cubes[j,move] = canonical_representative(tur
                end
                @info "NEW ROW" num (output_cubes[j],unique(output_new_cube
                num +=1
            else
                empty +=1
            end
        else
            empty +=1
        end
    end
    if empty == length(output)
        @info "Finished"
        return
    end
    output = output_new
    output_cubes = output_new_cubes
end
@info "Finished"
return
end
```

7.2. **Acknowledgements.** I would like to thank my Advisor Dr. Daniel Glasscock for working with me with my project, Aida KadicGaleb for being my honors comitee member for this project, Guido Dipierto for his initial code, and JuliaHub for their online computing.

References

[1] Doug Badger. *How to Solve a Rubik's Cube*. URL: https://codepen.io/igorgetmeabrain/full/MWjJEOy (visited on 04/22/2022).

[2] Robert G. Bryan. *Cube Loves God's Algorithm for the 2x2x2 Pocket Cube*. Apr. 12, 1993. URL: http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Jerry_Bryan__God's_Algorithm_for_the_2x2x2_Pocket_Cube.html (visited on 04/22/2022).

[3] Guido Dipietro. *2x2x2 Rubik's Cube modelled in Julia*. Sept. 2021. URL: https://github.com/GuidoDipietro/julia-small-rubiks-cube-model.

[4] A.H. Frey and D. Singmaster. *Handbook of Cubik Math*. Enslow Publishers, 1982. ISBN: 9780894900600. URL: https://books.google.com/books?id=SWIPAQAAMAAJ.

[5] Tomas Rokicki and Morley Davidson. *God's Number is 26 in the Quarter-Turn Metric*. URL: http://cube20.org/qtm/ (visited on 04/22/2022).

[6] Tomas Rokicki et al. "The Diameter of the Rubik's Cube Group Is Twenty". In: *SIAM Review* 56.4 (2014), pp. 645–670. DOI: 10.1137/140973499. eprint: https://doi.org/10.1137/140973499. URL: https://doi.org/10.1137/140973499.

[7] Tamar. *The devils number - history of the cube mystery*. May 20AD. URL: https://getgocube.com/play/devils-number/.

Department of Mathematics, University of Massachusetts Lowell, Lowell, MA 01854

Email address: 'Ryan_Brown2@student.uml.edu