Rubik's Cube Solvability Classes

Tulba-Lecu Theodor-Gabriel
Dragomir Ioan
email: [REDACTED]
Institute: [REDACTED]

March, 2020

Abstract

blah blah cube blah blah maths blah blah stupid.

Contents

1 Introduction

1.1 History

1.2 Notation

blah blah F F' and so on

1.3 Motivation

The motivation for this research came when the we were playing around with a Rubik's Cube. At some point, one edge piece flew away, and we put it back in the wrong orientation. Nevertheless we continued to solve the cube and observed that it was impossible to solve the cube.

We then asked ourselves the following question: *If we define a transition from one cube state to another, using only valid moves, an equivalence realtion, how many equivalence classes are there?*
We quickly amde some assumptions and worte some code to bruteforce all the states *add refference here*. We saw quite fast that the answer was 12, a result which corresponded both with our intuition and with other results *add refferences*. But where did this number come from? how does the answer change if we change the size of the cube?

The purpose of this paper is to answer those questions.

2 Solving the 3x3x3 Rubik's

blah blah summary of the aproach

2.1 Reduction to last-layer

blah blah proof by contradiction and deterministic solution

2.2 Naive full search space exploration

Our first approach consisted in exploring all possible last layer states and modelling the possible transitions in between these. In order to simplify the search, we abstracted away the notion of single turn moves, thus only focusing on PLL and OLL algorithms given their documented existence, but not needing to simulate their constituent moves. For instance, in the case of the H permutation, it is superfluous to simulate the move sequence [M2 U M2 U2 M2 U M2] and keep track of all 26 cubies, given that we know the final result will be that the first 2 layers remain unchanged and the last layer edge cubies keep their orientation but swap positions with their opposite. Thus, it is sufficient to only encode these position and orientation transforms last layer algorithms.

Through experimentation, we found that only a relatively minimal set of algorithms need to be implemented for full search space coverage. For example, all OLL algorithms can be reduced to a sequence of U turns and [F R U R' U' F'] sequences, and all PLL algorithms can be reduced to repeated U turns and Aa, Z, and Ua permutations.

States are encoded by 4 edge and 4 corner states. Edges and corners are numbered in counter clockwise order starting with the right edge and back-right corner, as illustrated in the following figure.

corner 1	edge 1	corner 0
edge 2		edge 0
corner 2	edge 3	corner 3

As a convention, we also number sticker colors according to what what edge they are on in the solved state. Color 0 is the color of the right face, color 1 is the color of the back face, etc.

$$edgeStates[i] = (color, flipped)$$
$$cornerStates[j] = (color, rotation)$$
$$i, j \in \{0, 1, 2, 3\},\ color \in \{0, 1, 2, 3\}$$
$$flipped = \begin{cases} 0, & \text{if the yellow sticker is facing up} \\ 1, & \text{if the yellow sticker is facing outwards} \end{cases}$$
$$rotation = \begin{cases} 0, & \text{if the yellow sticker is facing up} \\ 1,\ 2, & \text{by successive counter-clockwise rotations} \end{cases}$$

We define two operators on these states: flipEdge(edge) which flips an edge's orientation, and rotateCorner(corner, third_turns) which adds a certain number of third turns to a corner's rotation (addition modulo 3). By combining these two operators and permutations on the edges' and corners' positions we can achieve any last layer algorithm. For example, the Sune algorithm can be decomposed into the edge permutation,

$$\begin{pmatrix} edgeStates[0] & edgeStates[1] & edgeStates[2] & edgeStates[3] \\ edgeStates[2] & edgeStates[1] & edgeStates[3] & edgeStates[0] \end{pmatrix}$$

three corner rotations,

rotateCorner(cornerStates[1], 2)
rotateCorner(cornerStates[2], 2)
rotateCorner(cornerStates[3], 2)

and two corner swaps, represented by the permutation:

$$\begin{pmatrix} cornerStates[0] & cornerStates[1] & cornerStates[2] & cornerStates[3] \\ cornerStates[2] & cornerStates[3] & cornerStates[0] & cornerStates[1] \end{pmatrix}$$

To explore all possible states, we needed an efficient way to check whether a certain state was visited in the past. For this we created a simplistic hashing function which transforms a given last layer state to an integer, accounting for potential rotations of similar states. The hashing algorithm first rotates the cube so the 0 edge cubie is in the 0 position, and then employs a simple arithmetic coding (I might be way off and this might not be arithmetic coding at all) which multiplies an accumulator by 8 for edges and 12 for corners, adding the corresponding color and flip/rotation after each multiplication.

repeat
  RotateClockwise()
until $edgeStates[0].color = 0$
$hash \leftarrow edgeStates[0].flipped$
for $i \leftarrow 1, 3$ **do**
  $hash \leftarrow hash * 8 + edgeStates[i].flipped * 4 + edgeStates[i].color$
end for
for $j \leftarrow 0, 3$ **do**
  $hash \leftarrow hash * 12 + cornerStates[j].rotation * 4 + cornerStates[j].color$
end for
return hash

Now that we can quickly identify identical positions, we use an associative array whose keys are state hashes and whose values are integers so that two states have the same value iff it is possible to reach one state starting at the other, and vice versa.

We define a solvability class to be a maximal subset of the state set such that any two of its elements have a sequence of algorithms which take one state to the other. (this will probably be moved to some earlier section)

Thus, this array's values can be used as identifiers for the discovered solvability classes.

The search space exploration algorithm is as follows.

...

2.3 Giving an answer

2.3.1 Corner orientation

3

2.3.2 Edge orientation

2

2.3.3 Edge position

2

2.3.4 Final independence

3*2*2=12

3 Solving 4x4x4

blah blah analyse differences from 3x3x3

3.1 Generalizing theorems

easy to say, hard to do

3.2 Solving the symmetry problem

blah blah parity blah blah symmetric slices from the center

3.3 Solve the geometric invariant

blah blah probably said something dumb here

3.4 Results for 4x4x4

Hmmm...

4 Solving NxNxN

blah blah why not take it further

4.1 Prove independece of non-symmetric layers

probably true

4.2 Reduce to 3x3x3 and 4x4x4 cases

not to hard, maybe induction

4.3 Flex with vector spaces to calculate the answer

lol, mathy boiiiis

5 Generalizing to cuboids

blah blah even crazier idea

6 Solve the problem for other platonic solids

lol what is this even.