The Hardness of Solving an $n \times n \times n$ Rubik's Cube Optimally and Approximately

Zack Jorquera

December 11, 2020

1 Introduction

The Rubik's cube is a classic combination puzzle, invented in 1974 by Ernő Rubik, that utilized slices on a 3-dimensional cube that you can rotate to permute the stickers' locations on the cube. Traditionally, the Rubik's cube was only a $3 \times 3 \times 3$ cube with nine stickers of all the same color on each face (with six unique colors). It has always been considered a hard problem just to solve the $3 \times 3 \times 3$ Rubik's cube. Trying to solve the $3 \times 3 \times 3$ Rubik's cube optimally is an even harder problem, with the first algorithm doing so created in 1997 by Richard Korf [4]. Now, this challenging problem of solving a Rubik's cube has since been generalized to a whole bunch of different sizes, with larger sizes being harder to solve. These questions have lead people to ask how hard is it to solve an $n \times n \times n$ Rubik's cube optimally. We will look at this exact problem and look at how hard it is to approximate an optimal solution to a Rubik's cube.

The specific problem that we will be looking at is the problem of solving an $n \times n \times n$ Rubik's cube optimally. That is, given a configuration of stickers on an $n \times n \times n$ Rubik's cube, what optimal combination of turns, if any, will cause the Rubik's cube to go to a solved configuration such that there is no other combination of fewer turns that will also have the Rubik's cube go to a solved cube configuration. For this problem, we will be using the quarter-turn metric, which sees a single turn as a ± 90 degree rotation of a slice around one of the three axises. This is different from the more commonly used turn metric, which counts a 90-degree turn, a 180-degree turn, and a 270-degree turn (same as -90 degree), all as different individual turns.

To help study this optimization problem, we can define the associated decision problem, can an $n \times n \times n$ Rubik's cube be solved in k or fewer turns. In this project, we will look at how Erik Demaine et al. showed that this associated decision problem is NP-complete [2]. We will also implement a DPLL style algorithm to find the optimal solutions complete with heuristics to help with branch and bound functionality. Lastly, we will create an approximation algorithm to solve the Rubik's cube as efficiently as possible in polynomial time and analyze how hard it is to approximate the Rubik's cube. [1]

2 Notation

A Rubik's cube has six faces. Each faces lies on an axis (x, y, or z). We label the faces Up, Left, Front, Right, Back, and Down. Each face has n^2 stickers placed on the smaller cubes that we call the cubies. For the purposes of this project, we only care about the surface level cubies of which there are $n^3 - (n-2)^3$. On a $n \times n \times n$ Rubik's cube there are always 8 corner cubies (with three stickers each), $12(n-2)$ edge cubies (with two stickers each) and, $6(n-2)^2$ remaining center cubies (with 1 sticker each).

We can then permute the cubie positions and rotations (and thus also the sticker positions) by turning one of the cube's slices. There are two types of notation that will be used to denote a turn. The first is axis based; this is when we define the turn with the axis it is on, the slice index, and if the turn is in the positive

[1] All code can be found at `https://github.com/ZackJorquera/rubiks-cube-solver`.

rotational direction or not. The axis is the same as the rotational axis. The index is like the coordinate system with the origin being at the center (the 0th indexed slice is the middle slice, or for even-sized cubes, there is no 0th indexed slice). The index increase or decreases by one as you look at slices further out in the positive or negative direction, respectively. Note, we don't recognize the middle slice, index 0, as a valid turn. Finally, we then call a turn in the positive direction if the rotation vector (determined by the right-hand rule) points in the axis's positive direction.

The other type of notation is more closely related to the standard Rubik's cube turn notation; it is face based. The turn is defined by the face the slice is closest to (ULFRBD), the number of slices in is (with 1 being the outermost slice), and if the rotation is clockwise to the face or not (note, in my code, the number of slices in it is, starts at 0).

Take, for example, the move $m = x_1 \circ y_1 \circ z_1$ and apply it to a 3×3 solved cube, we would get the state shown in Figure 1. You might notice that the order of the turns is in reverse to how they were written. This is because each turn represents a permutation and the '$\circ$' is a function composition, which means they must be evaluated right to left or innermost to outermost. If using face-based notation, we would write U' F' L' where the order to apply the turns is the order written (left to right).

3 NP-completeness

We can show that deciding whether an $n \times n \times n$ Rubik's cube can be solved in k or fewer turns or less is in NP. That is, we can define a verifier that takes in the input cube configuration, a number k written in binary, and a polynomial sized witness that is the k or fewer turns needed to return the cube to the solved state.

The key to proving that this is in NP is showing that every configuration can be reached in a polynomial amount of moves relative to the size of the cube. This is called the "God's Number" [3] and for an $n \times n \times n$ Rubik's cube it is $p(n) = \Theta\left(\frac{n^2}{\log(n)}\right)$ which is polynomial. This means that our witness will always be at most $s = \min(p(n), k)$ turns needed to solve the cube, which means that verifying the solution by making at most s turns can be done in polynomial time (as there are $O(p(n))$ turns in the witness). This means that the problem of deciding whether an $n \times n \times n$ Rubik's cube can be solved in k or fewer turns is in NP.

To show that this problem is NP-hard, Erik Demaine et al. showed that they could reduce the NP-complete problem called the Promise Cubical Hamiltonian Path problem [2], to the Rubik's cube problem. In their paper, Erik Demaine et al. prove that the Promise Cubical Hamiltonian Path problem is, in fact, NP-complete; this is not something we will do here. The problem is defined as follows: given n length-m bit string $l_1, l_2, \cdots, l_n$ where $l_n = 00...0$ is the all-zero bit string we construct a graph where each node represents a bit string, and there is an edge between two nodes if their two bit-strings have a one Hamming distance separation. The decision problem is if there is a Hamiltonian path starting at l_1 and ending at l_n. In other words, is there some permutation of the bit-strings (keeping l_1 at the beginning and l_n at the end) so that every two adjacent bit-strings has a Hamming distance of one.

The reduction will consist of "encoding" the bit string into the Rubiks cube with some polynomial number of turns and then asking if the resulting configuration can be solved in fewer turns, specifically in $2n - 1 = \Theta(n)$ turns. The encoding for one bit-string, l_i, can be defined as a move b_i for a cube with size $s = 6n + 2m$. Specifically, we can define b_i as the following composition $b_i = (a_i)^{-1} \circ z_{m+i} \circ a_i$ where $a_i = (x_1)^{(l_i)_1} \circ (x_2)^{(l_i)_2} \circ (x_3)^{(l_i)_3} \circ \cdots \circ (x_m)^{(l_i)_m}$ where $(l_i)_k$ is the kth bit in the l_i bit string. Also, remember that x_i refers to making a turn on the index i x-axis slice in the positive direction. Note that because the turn index is always positive, we only really care about the positive half of the x-axis, which means that any turns on the negative half of the cube will not be optimal for a sufficiently large cube. We can also see that every turn in a_i commutes as they are all on the same axis.

One very important thing to realize about these b_* encoding is that for any b_i and another b_j, they will commute. They don't affect any of the same cubies if $i \neq j$ (if they are the same, then they commute trivially). b_i, for example, only affects the cubies along the z_{m+i} slice and the cubies on the Up and Down faces that lie on the $y_{-(m+i)}$ slice. b_j, on the other hand, only affects the cubies along the z_{m+j} slice and the

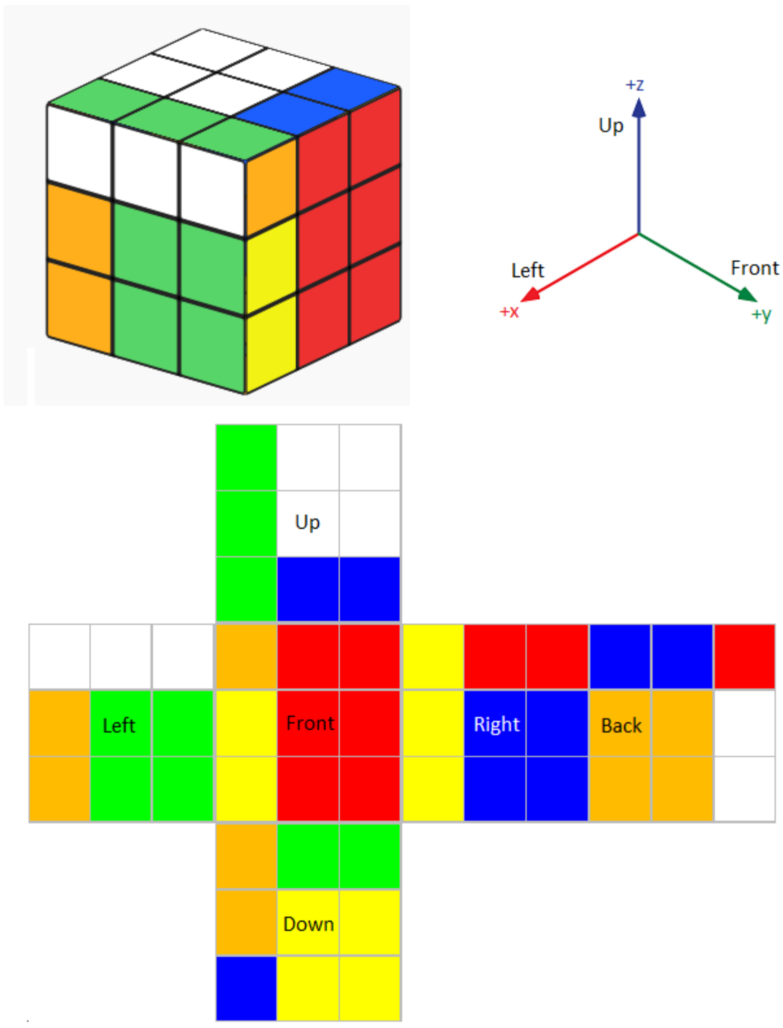

Figure 1: State after applying the move $m = x_1 \circ y_1 \circ z_1$ on a $3 \times 3 \times 3$ which in the order of the turns is U' F' L'. Note, U' means we are turning the up face counter clockwise as opposed to U which means we turn it clockwise.

cubies on the Up and Down faces that lie on the $y_{-(m+j)}$ slice. Note, the z slices (which exception to the outermost slice) do not cross the Up and Down faces. Because these two moves do not share any affected cubies, we can conclude that b_i and b_j commute.

Now, we create the full encoding of all n bit-strings, i.e. $b = b_1 \circ \cdots \circ b_n$. We also have to encode the starting bit-string so that the promise aspect of the problem is held, which would give us the final problem encoding of $t = a_1 \circ b$. Note, t requires $O(nm)$ turns. Finally, we ask the question: Can the Rubik's cube mixed up with move t be solved in $k = 2n - 1$ turns or fewer.

The idea here is that each turn along the x-axis correlates with flipping a bit when going from one bit-string to another. We can think of what x-axis turns have been applied as the current "bit-string state" of the cube. When the cube is fully solved, and also before we apply the final a_1 move in $t = a_1 \circ b$, we are in the all-zero "bit-string state." This should make sense for the solved state because we expect the all 0 bit-string to be the final bit-string in the Hamiltonian path (and thus the solved cube state). For the latter case, we apply a_1 so that we start in the l_1 "bit-string state." To change the "bit-string state" of the cube from a_1 to another a_j, we can apply x-axis turns. The minimum number of turns that we can apply is the same as the Hamming distance between the two bit-strings l_1 and l_j. Lastly, to undo one of the z turn moves, we have to be in the same "bit-string state" that the cube was in when the turn was originally made. Therefore, to return to the solved state, we have to "pass-through" each of the "bit-string states" and undo the z turn. Note, the b_i move first changes the "bit-string state" to be the a_i/l_i state, then applies the z_{m+i} move, and finally returns the cube back to the initial "bit-string state" which we assumed to be that of all zeros.

We can now show that if a solution to the Promise Cubical Hamiltonian Path problem exists, then a solution to the newly created Rubik's cube problem exists. We can do this by simplifying t and then taking the inverse. If we re-arrange the b_is in the same order as in the solution for the bit-strings in the Promise Cubical Hamiltonian Path problem then we will have $t = a_1 \circ b_1 \circ b_{k_2} \circ b_{k_3} \circ \cdots \circ b_{k_{n-1}} \circ b_n$ where $\{k_1, k_2, \cdots, k_n\}$ is the order of the bit-strings in the Promise Cubical Hamiltonian Path solution. We can then note that the a_1 and the a_1^{-1} in the b_1 cancel. Then for every remaining a_i and a_j^{-1} from every two adjacent $b_i \circ b_j$ we get most all turns to cancel except for the one turn. This is because the two bit-strings b_i and b_j are only one hamming distance away from each other, which means that the turns required to turn what is effectively a_i into a_j is just one turn. Also, the final b_n would only consist of the single z_{m+j} turn (is the associated bit-string is all 0). Adding everything up we the following simplified version of $t = z_{m+1} \circ (a_1 \circ a_{k_2}^{-1}) \circ z_{m+k_2} \circ (a_{k_2} \circ a_{k_3}^{-1}) \circ z_{m+k_3} \circ \cdots \circ z_{m+n}$. As mentioned above, every $(a_i \circ a_j^{-1})$ is only one turn. This would then leave us with exactly $k = 2n - 1$ turns, which means that the inverse of t, t^{-1}, is also less than or equal to k turns, which means that the Rubik's cube problem is successful.

If there is no solution to the Hamiltonian path problem, then the Rubik's cube's optimal solution has more than k turns. That is because when we try to find the solution the way we did above and simplify t, the optimal permutation, the b_is will lead to at least one $(a_i \circ a_j^{-1})$ that can't cancel out to be just one turn. This mirrors the fact that any permutation of the bit-strings without a valid ordering can not all be one Hamming distance one away from each other, as that would suggest that there is a solution.

Erik Demaine goes into a lot of depth to show that this simplified t^{-1} is the optimal solution [2]. However, I will not touch on that here.

To finish off the reduction, at least for the FP aspect of the problem, we can then extract the order of bit-strings in the Hamiltonian path from the indices of the z-axis turns of the solution in the order the turns were applied (or in the written order of the simplified t). And for each index of the z-axis turn, we can subtract out the m value (remember that we turned z_{m+j} for the jth bit string).

We can take as an example, the following bit-strings: $l_1 = 011, l_2 = 110, l_3 = 111, l_4 = 100, l_5 = 000$ (which has the Promise Cubical Hamiltonian Path solution of l_1, l_3, l_2, l_4, l_5). We can then construct the b_i moves which would make $b_1 = x_3^{-1} \circ x_2^{-1} \circ z_4 \circ x_2 \circ x_3$, all the way to $b_5 = z_8$. The simplification would then give us $t = z_4 \circ x_1^{-1} \circ z_6 \circ x_3 \circ z_5 \circ x_2 \circ z_7 \circ x_1 \circ z_8$ and then the solution would be $t^{-1} = z_8^{-1} \circ x_1^{-1} \circ z_7^{-1} \circ x_2^{-1} \circ z_5^{-1} \circ x_3^{-1} \circ z_6^{-1} \circ x_1 \circ z_4^{-1}$. To then convert this back to the Hamiltonian Path solution we look at each z_j^{-1} in t^{-1} in order of application (inner most permutation to outermost of the compositions) which gives us $(z_4^{-1}, z_6^{-1}, z_5^{-1}, z_7^{-1}, z_8^{-1})$. When we extract out the indices and then subtract out $m = 3$ we get $(1, 3, 2, 4, 5)$ which is the correct solution.

As shown in Figures 2, 3, and 4, we can see how the state of the cube changes as we add more parts to the t configuration.

In Figure 4

4 DPLL and IDA*

All code can be found at `https://github.com/ZackJorquera/rubiks-cube-solver`.

Now that we have shown the problem to be NP-complete, we can talk about implementing a solver that finds a solution in less than k turns. Generally, we can branch on each possible single turn up to the depth of $s = \min(p(n), k)$, where $p(x)$ is the polynomial that represents the god's number of the cube. The number of possible turns that can be made at each step is about $6n = O(n)$, as there are three axises with about n

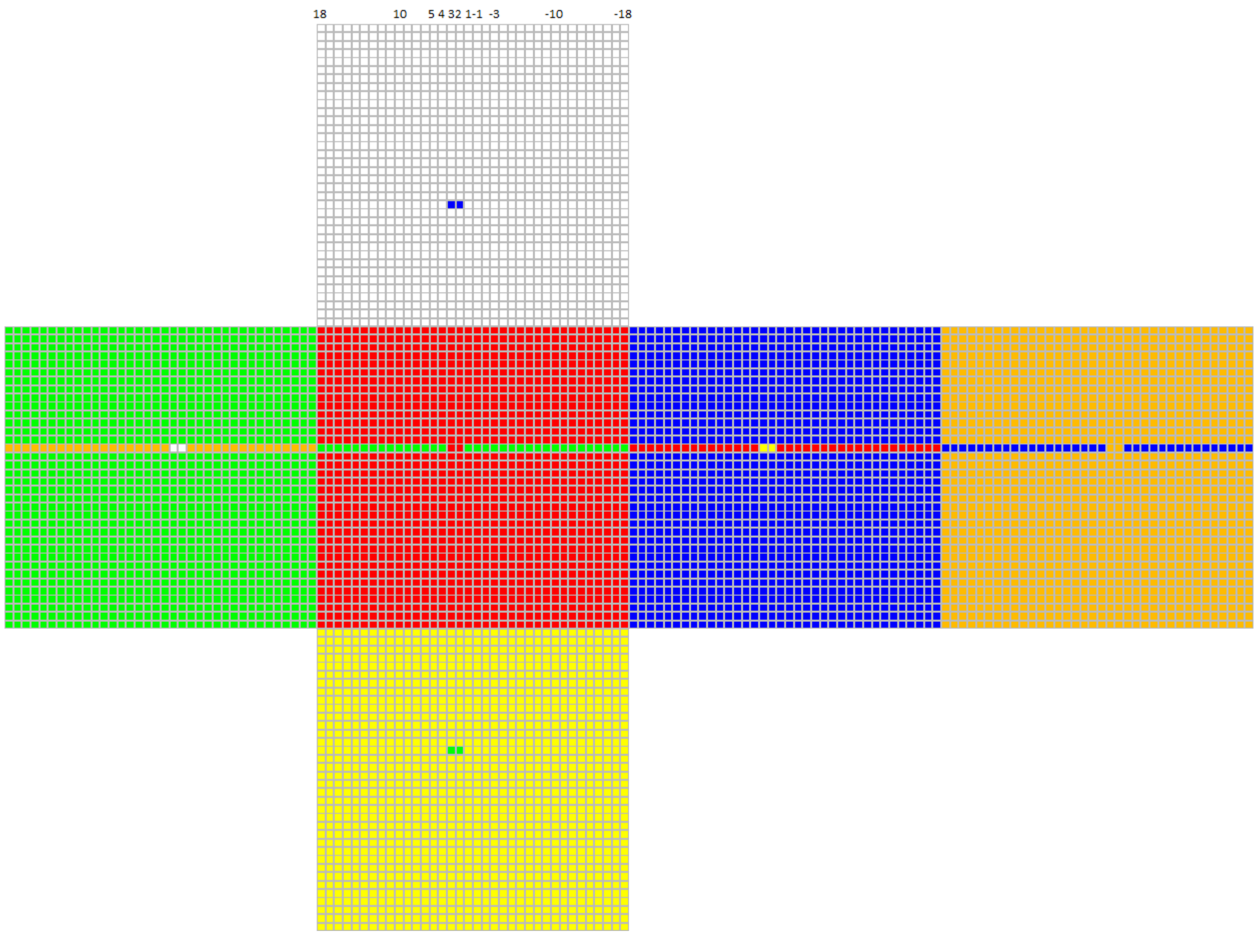

Figure 2: The state of the $36 \times 36 \times 36$ Rubik's cube after the $b_1 = x_3^{-1} \circ x_2^{-1} \circ z_4 \circ x_2 \circ x_3$ move. I also put the x-axis indices at the top.

slices each (ignoring the middle slice if it exists) of which we can turn in two way each. Putting everything together, this will give us a total DPLL runtime of $O(n^{O(k)})$. Note, we can make some simple optimizations to decrease the branching factor a little by not allowing some branches. For example, if the turn x_2 was made, we will not allow x_2^{-1} to be made next as that can never be in an optimal solution. We also won't allow the same turn to be made three times in a row as that is the same as the inverse $x_2 x_2 x_2 = x_2^{-1}$.

One last simple thing we can do to limit the number of redundant branches is to look for commuting turns and only allow one permutation of the turns. for example, take the following three turn move $x_1 x_3 x_2$. Because they are all on the same axis, we know they all commute with each other, which gives us $3! = 6$ valid permutations of turns for the same move. The easiest way to solve this problem is to allow only turns that commute to be adjacent to each other if the indices are in sorted descending order relative to the order they are applied to the cube. This would mean the only valid permutation of the move shown above would be $x_1 x_2 x_3$.

We can improve this DPLL algorithm in even more ways; namely, we can implement some heuristics and a branch and bound technique that uses them. There is a well-known algorithm for solving $3 \times 3 \times 3$ Rubik's cubes called Korf's algorithm [4] that gives some useful suggestions for different types of heuristics that can be used alongside what Korf calls an IDA* algorithm (iterative deepening A*). While they work well for

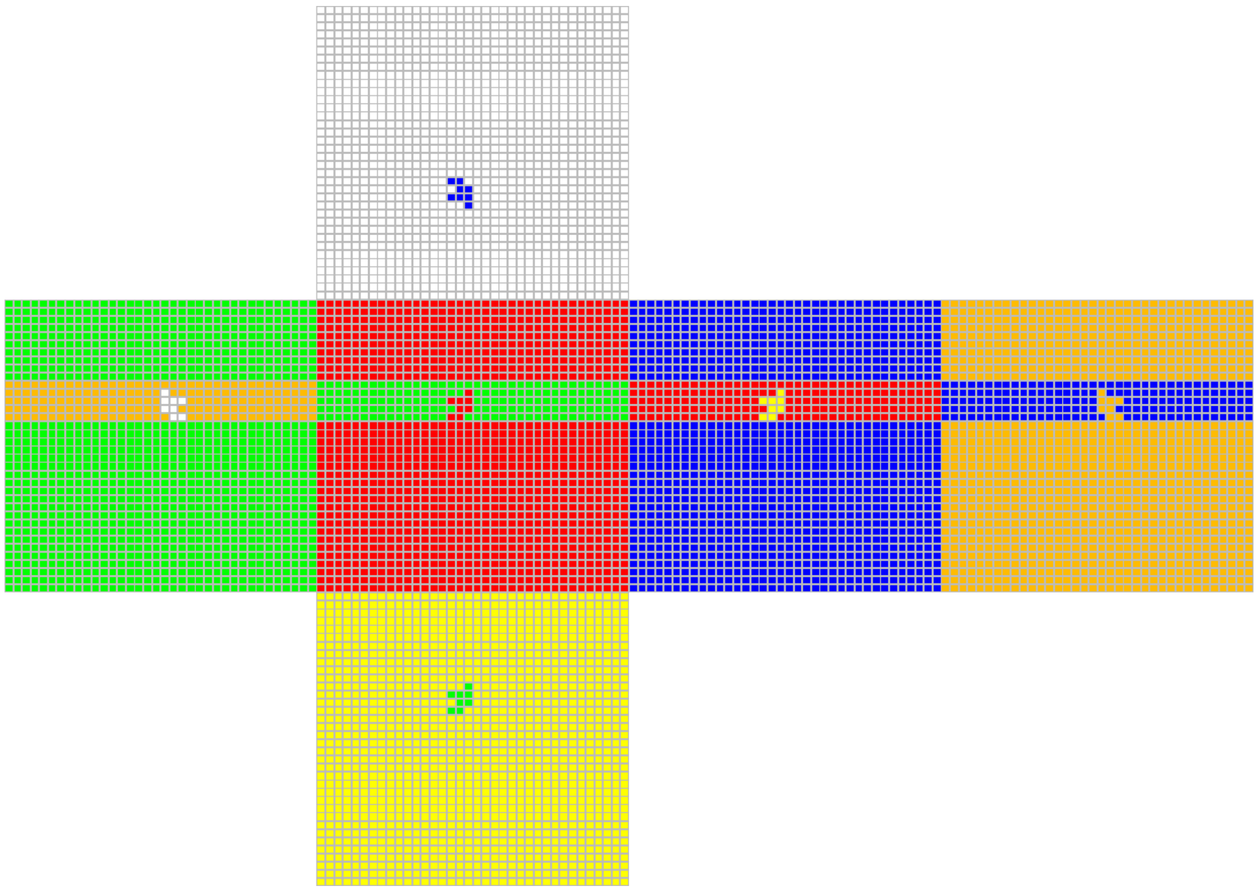

Figure 3: The state of the $36 \times 36 \times 36$ Rubik's cube after the $b = b_1 \circ b_2 \circ b_3 \circ b_4 \circ b_5$ move. You can notice how the bit strings have been encoded into the Rubik's cube where the bottom green slice (on $z_{m+1} = z_4$) on the Front face represents l_1 bit-string.

$3 \times 3 \times 3$ Rubik's cubes, these heuristics don't scale very well to $n \times n \times n$ Rubik's cubes, but they are the best we have. These heuristics/relaxations are just looking at the corner cubies and just looking at a subset of the edge cubies. These can be easily pre-computed, as they contain a small enough subset of states where the heuristics can be stored in a hash table in only a few gigabytes, which can then used in constant time, alongside a clever hashing algorithm, to help determine the best branch to look at first.

The first simple relaxation of the Rubik's cube problem that we will look at is to care only about the eight corner cubies and figure out how many moves it would take to "solve" them. This can be done easily as the eight corner cubies form a 2x2x2 Rubik's cube, which can then be solved in polynomial time. More so, we can pre-compute all of the different possible configurations and store the number of turns required for each configuration to be returned back to solved.

One beneficial optimization we can do for a 2x2x2 cube is to ignore half of the possible turns. For example, think about the move R L' (or $x_{-1} \circ x_1$), this is the same as the identity plus rotating the whole cube around the x-axis in the positive direction. Removing these non-moves is most easily done by restricting turns to only the positive faces (Up, Left, and forward). While helpful in most cases, this relaxation has problems; most notable, the solution to the 2x2x2 might not line up with the center cubies of the original cube as we consider any rotation of 2x2x2 the same. Thus, the solved corners' orientation would be defined

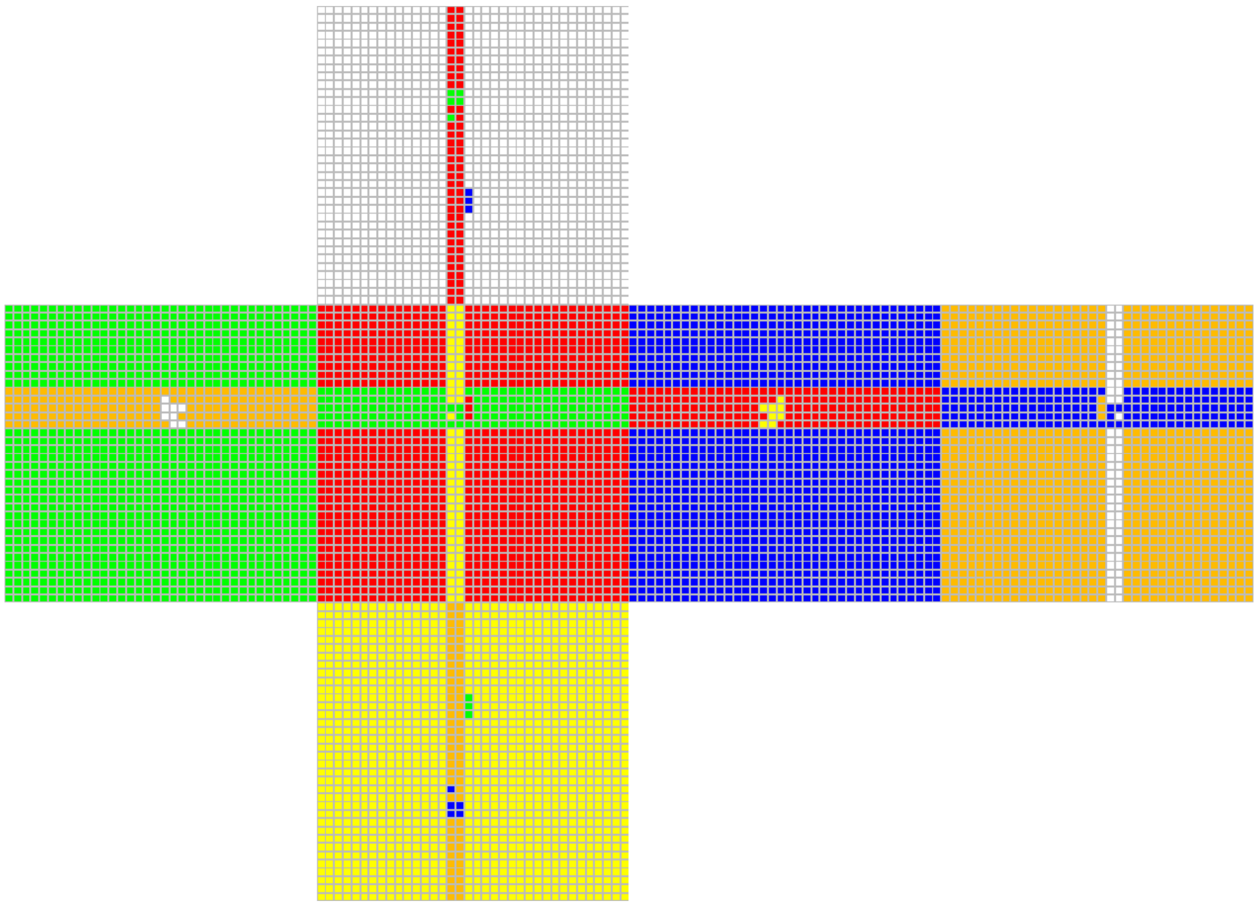

Figure 4: The state of the $36 \times 36 \times 36$ Rubik's cube after the $t = a_1 \circ b$ move, which is the final state we get from the reduction. You can notice that when the $z_{m+1}^{-1} = z_4^{-1}$ turn is made, we correct that bottom green slice. We can do this because we are in the l_1 "bit-string state," and if we try to correct a different slice with the appropriate z_{m+i}^{-1} turn we don't "correct" it, but instead mess up the cube more.

by what cubie was originally in the DRB (down-right-back) position. This, however, makes it significantly easier to pre-compute the turns need to solve a 2x2x2 for all of its distinct configurations as if we include orientation in the states, we would have about 24 times as many configurations.

For a 2x2x2 cube, there are $7!3^6 = 3,674,160$ different unique configurations when we don't account for different rotations. To make this true, we will require the cubie in the down, right, back position to be colored Yellow, Blue, Orange respectively (if it is not, we will rotate the cube first). All the different configurations can then be calculated and put into a hash table easily in the following way. We create a queue where we add the solved state of the cube. Then we start a loop while the queue is not empty. If the next state in the queue has not been added to the hash table, we will add it to the hash table and then add every state that is one turn away from the current state. Note, because we want to keep the DRB (down-right-back) cubie where it is, we will only make single turns from $G = \langle U, L, F \rangle$. The loop will end when there are no more states that haven't been reached. This ends up validating that there are, in fact, $3,674,160$ different unique configurations and that the "god's number" for a 2x2x2 Rubik's cube is 14.

We can now use this heuristic to determine if a branch is possible to be solved and also as a lower bound. For example, if we have a $n \times n \times n$ cube with $k = 15$ and we have already made six turns in our branch and this heuristic tells us that the corners require at least ten turns to be solved, then we know that there is no way to solve the cube in the nine remaining turns.

While it is suitable for small values of k, this heuristic can not help for any value above 14 as every valid $2 \times 2 \times 2$ cube can be solved in 14 turns or less. Also, as the cube's size increases, this heuristic's effectiveness completely disappears as more types of turns don't interact with the corner cubies. In general, all heuristics that we can use look at only a subset of the stickers, as for the cube to be solved, every subset must also be solved. For small subsets, like the corners, we can pre-compute a heuristic table to use in constant time. For larger subsets that are better for large cubes, we will have to create an algorithm to solve them optimally as any table would be too large to store in memory. This is problematic because, most of the time, the best algorithm is essentially DPLL. This suggests that we can create an optimized DPLL algorithm using a branch and bound technique to use on subsets on stickers that form a smaller cube, just like how the corners form a $2 \times 2 \times 2$ cube.

For a general $n \times n \times n$ Rubik's cube, we can pick a subset of stickers that functions as a smaller $m \times m \times m$ Rubik's cube if the stickers we pick are on the outermost slices of the cube. If we want m to be odd, we will also take the middles slice. Of course, this is only possible if n is odd and thus the original $n \times n \times n$ Rubik's cube has a middle slice. For example, if we have a $5 \times 5 \times 5$ cube, we can take the corners, the center edges, and the center face cubies, then we have a $3 \times 3 \times 3$ if we only move the outermost slices, which are also the only slices that can move these cubies anyways. Therefore we know we have a more manageable, solvable problem that can act as a heuristic. If we wanted to, we could keep chaining this on its self, for example, if we wanted to solve a $51 \times 51 \times 51$ Rubik's cube, we would solve a $49 \times 49 \times 49$ Rubik's cube, which would, in turn, solve a $47 \times 47 \times 47$ Rubik's cube and so on. This, while giving us a very accurate heuristic, would be very expensive to calculate as we would have to essentially solve the cube around $n/2$ times for one heuristic calculation (with each cube taking exponential time). Then for the next state we look at, we would have to calculate it all over again. Because of this, we will need to balance how effective the heuristics are versus how long they would take to calculate. That is, we want to make sure the majority of time is spent looking at the original cube size, not calculating the heuristics.

Now that we have heuristics defined, we can create a branch and bound type algorithm. The algorithm we will use is called Iterative deepening A* (IDA*) developed by Richard Korf [4]. To start this algorithm, we get the heuristics of the starting state of the cube. This will give us the starting depth that we will look for solutions of as we know there can not exist any solutions with fewer turns. From there, we will look at all moves that decrease the heuristic. More formally, we can define two variables g, the number of turns from the starting state, and h, the heuristic value that predicts the number of turns until solved. For every step, we only continue down the branch if $g + h$ is less than or equal to the current depth that we are looking for solutions. We can also sort which paths we follow to always minimize the $g + h$ value. Whenever we reach a branch that doesn't lead to a solution at the depth we are looking at, we can record the value of $g + h$ to be used as a future depth; more specifically, we care about the smallest $g + h$ we come across that is also greater than depth. Once all the branches at this depth have been explored, such that none lead to a solution. We can re-run the algorithm with the new depth of the smallest $g + h$ value we found.

This algorithm has a lot of advantages over basic DPLL. Namely, we will always find the optimal solution first. This is not a guarantee for DPLL. Also, we slowly increase the range that we look for the solution. With DPLL, we can at best naively choose a depth to look for the solution, and if it is too small, we will have to do it again, but if it is too large, then we will look through non-optimal turns, which will slow down how fast we can find the solution. The more we overshoot, the exponentially worse it is. However, if we know exactly how many turns the solution is, then the DPLL could be more efficient as it would not have to waste time looking for a solution at smaller depths. Currently, the implementation of the DPLL algorithm returns the first solution it finds. This is because it is designed to determine if a Rubik's cube is solvable in k or fewer turns.

On top of that, IDA* as it is, does no memorization. While this wouldn't be that hard to implement, it wouldn't be very practical. Every time we change the depth, we re-start looking for solutions at square one, essentially forgetting everything we learned previously. Also, it is possible for us to have multiple ways to get to the same state. Both of these suggest that it would make sense to calculate the heuristics and other information for each state. However, because of the enormous amount of different possible states that can exist for a Rubik's, it would not be feasible to store it all in memory. This doesn't mean we can't store data from only smaller depths. We can still create a hash table and only store heuristics if the g value is less than some value k. This will take $O(n^k)$ space.

We can then calculate timings for how long it takes to solve the Rubik's cube using DPLL and IDA* for different cubes' sizes and a different number of turns used to scramble the cube. We can see these timings in Figure 5. Also, the average time to compute the heuristics table is around 15.5 seconds.

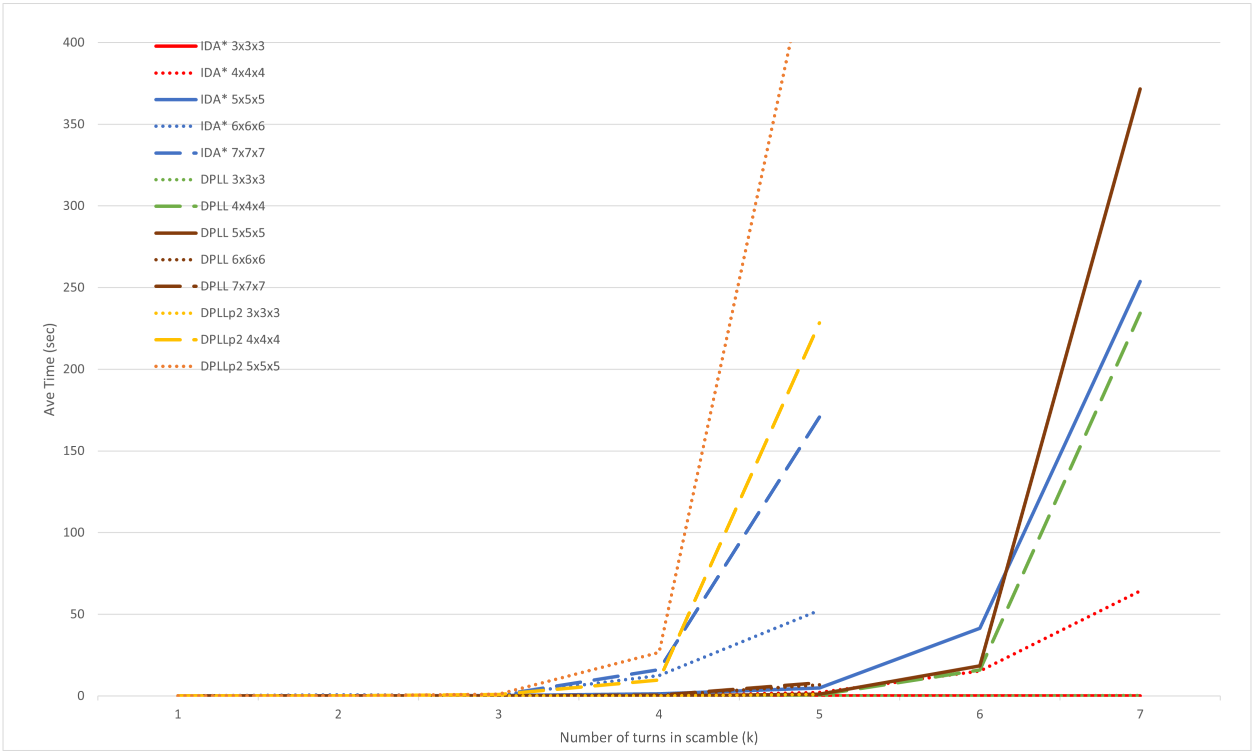

Figure 5: A plot of how long it takes an algorithm to solve a Rubik's cube. There are two algorithms, IDA* and DPLL, with the DPLL algorithm having two variants. One where the number of turns used to scramble the cube is known exactly (labeled as DPLL) and the other where the number of turns is overestimated by two turns (labeled as DPLLp2). Each data point is averaged over ten runs where every algorithm is given the same configurations.

We can now look at how the algorithms compare to each other. Generally, the time complexity of solving a Rubik's cube is $O(n^k)$. This is reflected in the plot as every line (where the cube size and algorithm type is constant) has exponential growth. If we look at the DPLL algorithm times, we can see some interesting stuff. First, overshooting the number of turns in a scramble (and thus the solution) brings a relatively large performance cost, which makes sense as the DPLL algorithm would naively search at the given depth. If we look at the DPLL algorithm, we can see that it is generally better than the IDA* algorithm until the end, when $k = 7$, for the smaller cubes ($n \leq 5$). This sort of makes sense as we would expect that the cost of looking through branches at a depth that the solution does not exist would give IDA* a disadvantage performance-wise. However, when the number of turns in the scramble gets sufficiently large, we see that this performance cost becomes less important; instead, the better pathfinding algorithm (A* for IDA*) gives us better performance overall at the depth with the solution in it, which is the most expensive. Also, IDA* uses a heuristic of $3 \times 3 \times 3$ cubes. In contrast, DPLL only uses the corner heuristic, which could also help give the implementation of IDA* an advantage. It can better determine when a branch does not have a solution at the correct depth in it.

5 Approximation

Like the Hamiltonian path problem we created for the NP-hardness proof, we can create another graph that can be used to determine the approximation complexity of the Rubik's Cube problem with an approximating preserving reduction. We will make this reduction from what we can call a Promise Cubical Travailing Salesperson problem. We will again be given n length-m bit string $l_1, l_2, \cdots, l_n$ where $l_n = 00...0$ is the all-zero bit string. We will then construct a graph where each node represents a bit string, and there is an edge between every node, with weight being the Hamming distance. We can then use this graph in a travailing salesperson problem where we start at l_1 and end at l_n.

This new Promise Cubical Travailing Salesperson problem is metric because if we were to define each node to be at the $\{0,1\}^m$ coordinate that its bit-string represents, the Hamming distance between two nodes is the same as the L1 norm of the difference of the two nodes. We know that the L1 norm satisfies the triangle inequality, i.e. for two nodes i and j, and an intermediary node k, $\|j-i\|_1 = \|j-k+k-i\| \leq \|j-j\|_1 + \|k-i\|_1$ and therefore this TSP is metric. When a TSP is in on a metric graph, then we know it is APX-complete with a no better than $\frac{123}{122}$-approximation [1].

Suppose we can create an approximation-preserving reduction from this metric Promise Cubical Travailing Salesperson problem to the Rubik's cube problem. In that case, we can show that Rubik's cube can't be approximated bellow an α factor approximation because otherwise, that would imply that PCTSP can be approximated better than its best-case approximation.

This reduction will be very similar to the NP-hardness reduction proof one. We create a cube with a sufficiently large size and then for each bit-string l_i we create the encoding b_i as $(a_i)^{-1} \circ z_{m+i} \circ a_i$ where $a_i = (x_1)^{(l_i)_1} \circ (x_2)^{(l_i)_2} \circ (x_3)^{(l_i)_3} \circ \cdots \circ (x_m)^{(l_i)_m}$ where $(l_i)_k$ is the kth bit in the l_i bit string.

Then, like for the NP-hardness reduction, we create the full encoding of the bit-strings of interest, i.e. $b = b_2 \circ \cdots \circ b_{n-1}$. Note that for the NP-hardness reduction, we included b_1 and b_n because we know that the optimization would force those to be first and last, respectively. This is not something we can assume here. Thus, we will basically assume that these are in the "solution" and preemptively apply them. This, however, does not mean we don't also have to encode the starting bit-string so that the promise aspect of the problem is held. This will give us the final encoding of $t = a_1 \circ b$, which takes $O(nm)$ turn. Then we ask to minimize how many turns are needed to solve the Rubik's cube mixed up with the move t. In the end, this only removes two z_{m+i} turns as the original a_1 cancelled out with the a_1^{-1} in the b_1 leaving just $z_{m+1} \circ a_1$ and the b_n move was just z_{m+n}. This makes out optimal solution two turns fewer than with the NP-hardness reduction. To avoid confusion, we will denote this reduction with r_2.

If we have a Promise Cubical Travailing Salesperson problem instance x, with n bit-string, with the optimal solution's weight being $\text{OPT}_{\text{PCTSP}}(x) = w_{\text{opt}} \geq n-1$. Then we can say that the optimal solution to the Rubik's cube problem is then $\text{OPT}_{\text{Rubiks}}(r_2(x)) \leq w_{\text{opt}} + n - 2$ turns. In fact, for any solution to PCTSP with weight w, there will always exist a solution to the Rubik's cube problem with $w+n-2$ number of turns. Remember that we will always have $n-2$ z_{m+i} type turns (this accounts for the two removed ones). For the most part, these z_{m+i} type turns are unrelated to the number of x-axis turns, which directly correlates to the weight of the Promise Cubical Travailing Salesperson problem solution's weight. This is because going from one "bit-string state" to another takes the Hamming distance between the two number of turns on the cube (which is the same as the edge weight between them in the graph). Therefore, we can always guarantee that there is a solution with $w+n-2$ turns.

We can do an approximation preserving reduction for any instance I_1 of the Promise Cubical Travailing Salesperson problem in the way defined above to get the instance $I_2 = r_2(I_1)$ of the Rubik's cube problem. For the reduction, let's define the promise Cubical Travailing Salesperson problem's objective function to be $n-2$ plus the actual weight. Let's call this new problem PCTSP2 to avoid confusion. The reduction from PCTSP2 to the Rubik's cube problem is still the same reduction as for PCTSP; however, we can now say the following.

$$\lim_{n \to \infty} \mathrm{OPT}_{\mathrm{Rubiks}}(I_2) \leq w + n - 2 = \mathrm{OPT}_{\mathrm{PCTSP2}}(x)$$

Note, we can not say that $\mathrm{OPT}_{\mathrm{Rubiks}}(I_2) = w + n - 2 = \mathrm{OPT}_{\mathrm{PCTSP2}}(x)$. I assume that this is the case for some sufficiently large cube so that for any optimal solution all other turns are restricted (especially face turns). This, however, I am unable to prove.

We now want to prove the other half of the approximation preserving reduction hold. That is, we need to define some function g such that for any solution s to the Rubik's cube problem instance created by the reduction, $I_2 = r_2(I_1)$, we can create a solution, l, to the original problem with $l = g(I_1, s)$. Using that, we then need to show that the following is true.

$$\mathrm{obj}_{\mathrm{PCTSP2}}(I_1, l) = w + n - 2 \leq \mathrm{OPT}_{\mathrm{Rubiks}}(I_2, s)$$

We will have to define g is two parts. First, if the solution takes the form of some permutation of the b moves, with only polynomial time simplifications done like for example when $e = m \circ m^{-1}$. Then we are left with a solution that is equivalently $t^{-1} = b_{k_{n-1}}^{-1} \circ b_{k_{n-2}}^{-1} \circ \cdots \circ b_{k_3}^{-1} \circ b_{k_2}^{-1} \circ a_1^{-1}$. We can extract out the z_{m+i}^{-1} values in the same we did for the NP-hardness proof to get the ordering of the bit-strings. Note, because we removed z_{m+1} and z_{m+n}, we will have to add l_1 and l_n to be first and last nodes in our solution to the PCTSP2 instance. With that we have our solution, $l = g(I_1, t^{-1})$. In this case we know that $\mathrm{obj}_{\mathrm{PCTSP2}}(I_1, l) \leq \mathrm{OPT}_{\mathrm{Rubiks}}(I_2, t^{-1})$ by definition.

The second case is for all other circumstances. The easiest to deal with is when the number of turns is greater than the max weight of the PCTSP2 problem. We can't calculate the max weight exactly, but we can pick a random permutation, which will be less than or equal to the max weight solution. For the case when the number of turns is less than the max weight of the PCTSP2 instance, and the Rubik's cube solution can't be easily transformed into some permutation of the b moves, we can't say anything about how the function g would work. I suspect that for some sufficiently large cube, just like for the optimal case, there doesn't exist any solution that can't be turned into some permutation of the b moves just by making polynomially computable simplifications of which we can extract the PCTSP2 solution. This, I am also not able to prove.

I suspect this is the case because the majority of the complexity comes from turning the outermost face slices. Demaine says this "[allows] rows of stickers to align in several directions over the course of a solution" [2, p. 11]. Therefore, we might be able to make it "cost" more turns to turn the faces slices and also solve to the cube so that we can easily "detect" it. We can think about what we consider to be a solved cube. All the stickers on the same face are all the same color; we don't care about where each specific cubie is. Therefore, it is possible to start with a solved state C_0, apply t to it, and then use the solution s, which gives us a solved state C_1 that has cubies in different locations than C_0. This would be a perfect example for when the solution l might not be able to be created from the function $g(I_1, s)$ as $s \circ t$ would not be the identity move (even though it results in a solved state). For example, on a 5×5 Rubik's cube, the move (U' 2R' F' 2L F 2R F' 2L' F U 2R' F' 2L F 2R F' 2L' F) will result in a solved state given that it starts in a solved state. However, the cubies are not in the same spots (this is illustrated in Figure 6). This particular move would most likely not affect the bit-string cubies and could probably be easily accounted for so that we can extract a b permutation solution. It is unclear how a move like this would be incorporated into a solution that can't be easily accounted for in a general case. It is furthermore unclear if these solutions would scale with the size of the cube.

This all make me conjecture that the approximation complexity of solving a $n \times n \times n$ Rubik's cube optimally is APX-complete. It might be possible to prove this is true using a gap preserving reduction where we first need to determine that gap-PCTSP is NP-hard for some known gap. This will be left up to further study.

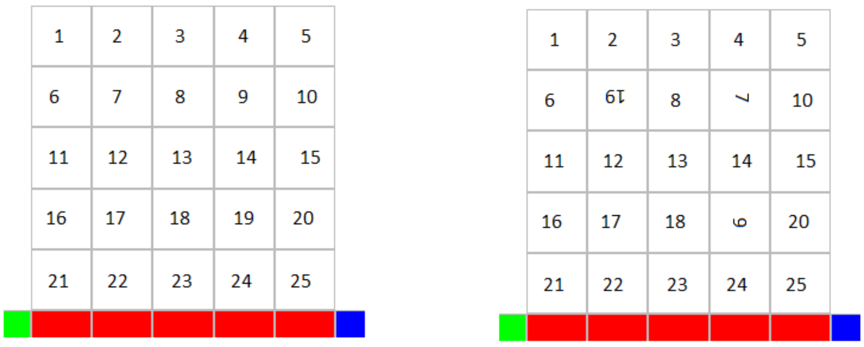

Figure 6: The state of the $5 \times 5 \times 5$ Rubik's cube before and after the move (U' 2R' F' 2L F 2R F' 2L' F U p 2R' F' 2L F 2R F' 2L' F) or $y_2^{-1} \circ x_1 \circ y_2 \circ x_{-1} \circ y_2^{-1} \circ x_1^{-1} \circ y_2 \circ x_{-1}^{-1} \circ z_2^{-1} \circ y_2^{-1} \circ x_1 \circ y_2 \circ x_{-1} \circ y_2^{-1} \circ x_1^{-1} \circ y_2 \circ x_{-1}^{-1} \circ z_2$ in permutation axis notation. On the left we have the original state, C_0, and on the right we have the resulting state C_1. To illustrate how the cubies move around, I drew numbers on the stickers. Note, even though I only drew the top face, cubies on the other faces move around too.

References

[1] Nicos Christofides. "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem". In: 1976.

[2] Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. "Solving the Rubik's Cube Optimally is NP-complete". In: *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)*. Ed. by Rolf Niedermeier and Brigitte Vallée. Vol. 96. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 24:1–24:13. ISBN: 978-3-95977-062-0. DOI: `10.4230/LIPIcs.STACS.2018.24`. URL: `http://drops.dagstuhl.de/opus/volltexte/2018/8533`.

[3] Erik D. Demaine et al. "Algorithms for Solving Rubik's Cubes". In: *Proceedings of the 19th European Conference on Algorithms*. ESA'11. Saarbrücken, Germany: Springer-Verlag, 2011, pp. 689–700. ISBN: 9783642237188.

[4] Richard E. Korf. "Finding optimal solutions to Rubik's Cube using pattern databases". In: *In Proceedings of the Fourteenth National Conference on Artificial Intelligence*. 1997, pp. 700–705.