Various Computational Cube Theory Conjectures

Stephen Huan

July 6, 2019

1 Basics

1.1 What This Is

This lecture will not directly make you faster. Nor is it really about Rubik's cubes. Instead, it is about general computer algorithms that currently exist and I postulate will exist. If you appreciate listening to someone talk about cubes, I encourage you to write your own lectures and make Rubik's Cube Club great again. In the words of Jayden McNeill, "Don't be a cubing intellectual".

1.2 Representation

At first, I imagined doing a sticker-wise face respentation. That is, an array of shape 6x3x3. The first dimention represents the face (there are 6 faces on a cube). Note that I specify a face by color in WCA orientation (white top green front) and colors by their first letter (they are distinct); the ordering is WGRBOY: a counterclockwise-esque rotation pattern. Then, for each face, there are 9 stickers. I then subdivided these 9 stickers into 3 rows of 3. The problem is that simulating a turn becomes too unweildy. A single move will shift 12 stickers along its axis, which are hard to keep track of, and shifts 9 stickers which are easier to cycle. In total 21 stickers move, and it's hard to generalize for the 6 possible moves.

An obvious alternative I realized when reading Kociemba's GitHub is a cubie-level representation, which simplifies moving. In my implentation a cubie contains 3 ordered colors. Cubie(W, B, O) means the white is either up/down (U/D), the blue is front/back (F/B), and the orange is left/right (L/R). In general, I also refer to moves by the equivalent face they are done to. The ordering is a representation of the concept of "orientation". The logic is the same for edge pieces. For example, Cubie(W, B, None) is a valid edge. The white center would be Cubie(W, None, None) and the even core of the cube would be Cubie(None, None, None).

As you can see the same class can represent each piece type. Now let's simulate a move. First, track a corner; in this case the WBO corner. When you turn the U layer the W remain facing U/D, but the O faces F/B and the B faces L/R. Now track the WGR corner as I do a R move. R remains facing L/R, but W now faces F/B and G faces

U/D. Suppose U/D is 0, F/B is 1, and L/R is 2. A move will not affect its corresponding color, and will swap the two remaining colors.

```
def rotate(self, dir):
    i, j = {0, 1, 2} - {dir}
    self.colors[i], self.colors[j] = self.colors[j], self.colors[i]
```

Now that I know how moves affect cubies, I just need to move the cubie objects themselves. First of all, I will make an array to hold cubie objects, this time in layers. I started with the top layer, going from left to right, top to bottom. This array has shape 6x9, as I felt it was no longer necessary to subdivide further into rows. To do a move, get the layer it corresponds to. Then, cycle the corners and cycle the edges. Finally, apply the rotation algorithm shown above on each piece in the layer. Note that when I "get" a layer I return indices, not the objects themselves. I need to modify the array, and since Python doesn't have pointers only indicies will do. Getting these indices is remarkably easy.

Here is my cycle algorithm. Note that it is extremely general, as I might change the dimentionality in the future.

```
def cycle(l, indexes):
    start = access(l, indexes[-1])
    for i in range(len(indexes) - 1, 0, -1):
        modify(l, indexes[i], access(l, indexes[i - 1]))
    modify(l, indexes[0], start)
```

Finally, here is how I perfrom a move. Note that CWC and CWE are lists of indices specifying the order.

```
def move(self, dir):
      layer = self.get_layer(dir)

      cycle(self.cube, get(layer, CWC))
      cycle(self.cube, get(layer, CWE))

      for cubie in layer: access(self.cube, cubie).rotate(ORIENT[dir])
```

Some details: I specify color using numeric codes, because:

1. The color ordering is now WYGBRO.

2. The opposite color to a given color c is $5 - c$, like dice.

3. The orientation of a move is $j//2$, where j is the index of the move in the ordering.

Of course, you could use strings or whatever to your liking. It doesn't really matter.

1.3 Commentary

Evidently, a cubie level approach is superior to a sticker level approach. I have some intuition as to why.

- Traditional speedsolving methods emphasize blockbuilding and a piece level approach. We like to make fun of beginners for saying "I can only solve a face", as it shows a fundamental näivety with the geometry of the puzzle.
- In general, the more abstract something is, the easier it is to manipulate.

If you look at the structure of my code, avaiable here you'll realize I like named constants, small, modular, functions, and headers. I call this "AI style". I do not write production code. As such, I think this format is good enough for a beginner playing around at programming. As a side note, you, the reader, should take AI. It is by far the greatest class TJ has offered me in these two long and stressful years and has provided a foundation that will last me the rest of my career. This lecrure would not be possible otherwise.

2 High Level Algorithmic Overview

2.1 Problem Structure

Given a cube state, the problem is to find the shortest move sequence such that the cube is solved after the sequence. That is, I want to solve a cube in the fewest moves possible. This problem is a graph problem. The vertices are states of the cubes, and the edges are possible moves. Obviously this graph has cycles, but we can prune those out. Now, this is a single-source-shortest path problem, on an undirected graph.

2.2 Uninformed Searches

2.2.1 BFS

Imagine copying the cube 18 times and then applying U to one of the copies, R to another, etc. until I have 18 distinct cubes. Then for each of those cubes, I do the same thing. If I enumerate every single possibility, I will of course find the shortest solution. This algorithm is called breadth-first-search or "BFS" and it has the computational complexity of $O(|V|+|E|)$ as in the worse case I visit each vertex and each edge exactly once. In this case, since, the graph is a tree, $|E|=|V|-1$. Therefore the complexity is $O(|V|+|V|-1)=O(|V|)$.

That seems pretty good, as it's linear in $|V|$. Unfortunately, the number of vertices grows at an exponential rate. It's more accurate to say the complexity is $O(b^d)$, where b is the branching factor, the number of moves at each depth, and d is the depth.

If I'm in half turn metric (HTM) then there are 15 (6*3, but minus 3 since it doesn't make sense to do a move and then a move on the same face) possible moves at each step. God's number is 20, so in the worst case I'd have to search 15^{20} states (approximately $3.3 \cdot 10^{25}$ states) which is way too large. Clearly, I need a faster algorithm.

2.2.2 Bi-BFS

A simple, but enormously effective optimization. Do a BFS, but from start to goal and goal to start simultaneously and symmetrically. When the two searchs meet in the middle, just combine the two paths. Intuitively, two small trees will halt the exponential growth in half. When I solve mazes, I'll jump from end to end and try to combine the two. The new complexity is $O(b^{d/2}) = O(\sqrt{b^d})$, doubling my effective search depth.

Some subtleties in implementation: adding children to seen is much faster, but don't goal test on children. My implementation is extremely slow though: it takes 30 seconds to find the optimal 9 HTM U-perm.

2.2.3 IDDFS

Essentially the same thing as BFS but uses less memory. Since DFS goes straight down, only the path from start to current node needs to be stored. Therefore, the memory consumption is $O(d)$. The trade off is time, as iterative deepening means running DFS to depth 0, then 1, then 2, etc. until depth d. The complexity is $O(1 + b^1 + b^2 + b^3 + ... + b^d)$. Using geometric series this sums to $\frac{b^{d+1}-1}{b-1} = \lim_{b\to\infty} b^d$, because of the definition of big-O (asymptotic).

An addded benefit is it avoids the copying cost, which is a nontrival factor (#1 in total time when profiled). Instead of copying a cube object to generate children, just stick with one cube object and perform a single move when going down, and doing a single inverse move when going back up. With a BFS, I would need to apply multiple inverse moves, making it slightly slower than just copying.

A comparision of all the algorithms discussed so far:

Algorithms

	BFS	Bi-BFS	IDDFS	Bi-IDDFS
Time	$O(b^d)$	$O(b^{d/2})$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{d/2})$	$O(d)$	$O(d)$

2.3 Informed Searches

We can do better. If I take advantage of a prediction of how far away a given state is to the goal, then the search is faster. The general algorithm for this is called A*, and even though it uses a heuristic, the search will return an optimal path if the heuristic is admissible. Optimality proof TBD.

3 What Exists

The undisputed best algorithm for solving 3x3 is, of course, Kociemba's algorithm. It works by some horrendously complicated two-phase group reduction math stuff. A "group", as I understand it, is a subset of cube states able to be solved by a subset of moves.

Drawbacks:

- Not generalizable: 4x4+ group reduction would be intractable (necessary to do slice moves)
- Fundamentally a search algorithm: Stockfish was made a decade ago.
- Takes ˜12 seconds to solve a cube optimally.

4 The Future

Now, there exists machine learning. ML traditionally deals with vectors, so I propose one-hot encoding colors and then flattening into a linear order, sticker by sticker face by face. The resulting dimentionality would be 6*9*6 = 324x1. The reason I don't use numberical color codes is because that causes numerically similar colors to be literally closer to each other, which is something I don't want. Therefore, I make the colors orthongonal to each other.

4.1 Pure Prediction

The most direct approach would be to have a neural network (NN) or convolutional neural network (CNN) directly predict the next move given a cube state. The problem would be probable difficulty in extracting a viable feature space and I have no idea how good this would actually be. Also, there'd be no way to verify optimality on a input that hadn't been solved before.

4.2 Boosting A*

I mentioned earlier that A* required an admissible heuristic. The deterimination of such a function is nontrivial and NNs could simulate a heuristic. The problem would be the lack of guarentee of admissibility, which is solved by the following approach. First off, add a regularization factor in loss to penalize overestimation more than underestimating. This by itself does not guarentee admissibility, thought it encourages it. Suppose the heuristic is y' and the actual distance is y. I want a function f such that $\forall y' f(y') \leq y'$ and $f(y') < y$. That is, if a function satisfies both conditions, then it must be admissible. A "trivial" function that accomplishes this is $f(y') = y' - e$, where e is $\max_{\forall y'} y' - y$. However, this linear "force" layer may be too aggressive, as I want the heuristic as large as possible. Stack NNs on top of each other sucessively to soften the blow, then add the force layer to force admissbility.

4.3 Reinforcement Learning

In my opinion, this has the greatest viability. AlphaGO is good example of RL's power. The combination of a small discrete action space coupled with a massive state space is very comparable to Chess, Othello, Go, etc.

The added benefit is the Rubik's cube is arguably much easier than Chess or Go, which have probably 5x the action space and depths in the hundreds to thousands. Such a system would be extremely general as applying it to 4x4 is as easy as generating more training data.