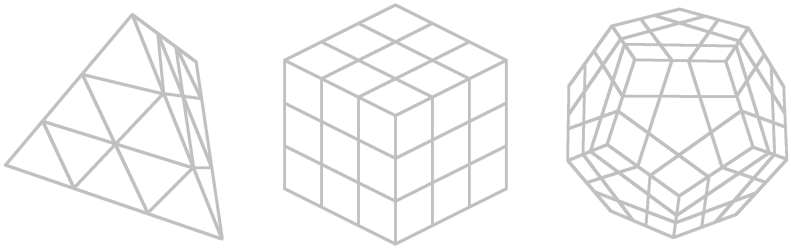
Diploma Thesis

Analyzing Human Solving Methods for Rubik's Cube and similar Puzzles

Stefan Pochmann



Supervisor
Prof. Dr. J. Fürnkranz

March 29, 2008

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, März 2008 Stefan Pochmann

Contents

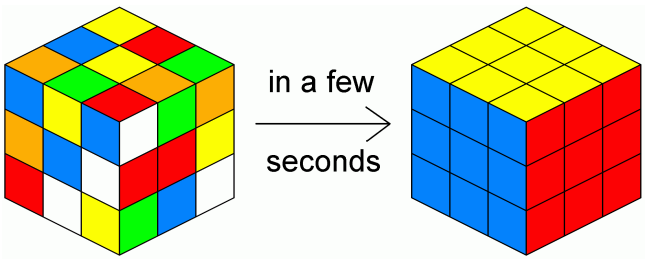
List of Figures

List of Tables

Foreword

Rubik's Cube became a worldwide phenomenon in the early 1980s, puzzling young and old alike. At first it seemed nearly impossible to solve, but soon people invented solving methods and many were published in magazines or as books. Competitions arose in which people tried to solve the cube in the shortest time, culminating in a world championship in 1982 with competitors from 19 countries. Soon after this, the cube's popularity faded quickly.

For many years, new puzzles similar to the original cube kept emerging from time to time, and fans around the world still existed. But it wasn't until around 2002 and internet discussion forums that they became aware of and connected to each other. In 2003, the first official championships after 20 years were held, and the year 2007 saw 53 official championships around the world.



The solving times have improved a lot over the years. The world champion of 1982 took 22.95 seconds in his best of three attempts, now the world record is 9.18 seconds for a single solve and 11.33 for an average of five solves. There are three main reasons for this: Better cubes, more highly devoted people, and better solving methods. This diploma thesis deals with that third aspect, solving methods.

Several different methods are used to solve the cube, all working the same basic way. Each starts by solving a certain few pieces, adding some more, and so on until the cube is fully solved. For example, the typical begin-

ner method starts by solving the top layer one piece at a time, adding the middle layer one piece at a time, then the bottom layer in four steps.

Computer programs which can solve the cube already exist. Some implement a fixed human method, others use general search algorithms to find shortest solutions possible. In most cases the cube can be solved in 18 turns, and some programs are able to find these very short solutions. Optimal solutions however are incomprehensible to humans. At best they can be somewhat followed, but humans aren't able to repeat this feat except with specialized methods, a lot of time for trial and error, as well as some luck. The fastest human solvers usually use around 50 to 60 turns.

This diploma thesis describes the development and results of a flexible computer program for analyzing human solving methods. The idea is to let someone explain any method to the computer in a simple way, just specifying the subgoals on the path to the final goal of a fully solved cube. The program then finds the actual turns to reach the subgoals and provides statistics to evaluate the quality of the method, as well as sample solves that the user can then study in detail. The overall purpose thus is to test new method ideas quickly without having to do the dirty work, judge methods to find the best, and learn from the found solutions. Also, while this thesis covers only the standard cube, the program is actually designed to handle other puzzles as well. For this, the puzzle is described by the user as part of the input.

Overview

The remainder of this thesis is organized as follows:

Chapter 1 describes the Rubik's cube, introduces the cubing community, shows some human solving methods and observes what they have in common.

Chapter 2 discusses several previous programs for solving the cube and what they're good at, and results concerning the "hardest" cube states.

Chapter 3 explains why those previous programs are not suited for method evaluation, and introduces the Hume program which addresses exactly that.

Chapter 4 uses Hume to compare solving methods for Rubik's cube, discussing statistics and solutions computed by the program.

Chapter 5 describes the implementation of Hume, particularly the search for solutions.

Chapter 6 discusses possible future developments and applications of Hume.

Chapter 7 offers some conclusions.

Appendix A shows counting arguments proving that some cube states require 18 turns to be solved.

Acknowledgments

I want to thank my supervisor professor Fürnkranz, especially for encouraging me to use my hobby and my expertise in it for my diploma thesis. My gratitude also goes to the worldwide cubing community for the fun, the friendly competition, and the interesting discussions I enjoyed with cubers in the last few years.

Accompanying website

Note that this diploma thesis uses many pictures of puzzles which are colorful in nature. It's therefore advisable to read it in color. The original version is available on the web, together with the Hume program and clickable links for the references.

`http://www.stefan-pochmann.info/hume/`

Chapter 1

Cube basics and solving

Rubik's cube is a mechanical puzzle, a colorful cube made up of 3*3*3 smaller cubes held together by an internal mechanism that allows rotating layers of the puzzle[1]. Randomly doing so leads to the cube getting scrambled, and the goal is to turn it back to the solved state where each of the six sides shows only one color.

This chapter introduces the speedcubing community, describes some basic properties of the cube, shows some human solving methods and analyzes what they have in common.

1.1 Speedcubing

There's a worldwide very active community of people devoted to not just solving the cube at all, but as fast as possible. This is called "speedcubing", and people doing it are called cubers or speedcubers. I myself am a speedcuber as well, having joined the community in 2003.

Since that year there have also been many competitions all around the world. In 2003 there were just two, but since then the number has increased every year, with 53 competitions in 2007 and already 21 in the first three months of 2008. Besides solving the standard Rubik's cube the standard way, we also solve several other puzzles (see figures 1.1 and 1.2), and solve them one-handed, blindfolded, with feet, or with fewest turns. My favorite puzzle is the Megaminx, with which I've been world champion and world

[1] It's not really made up of 3*3*3 smaller cubes, but that's what it behaves like and the actual mechanism is irrelevant here.

record holder. I'm also very interested in solving blindfolded, invented several methods for it which have become quite popular, and was the first to solve a 5x5x5 cube blindfolded in a competition.

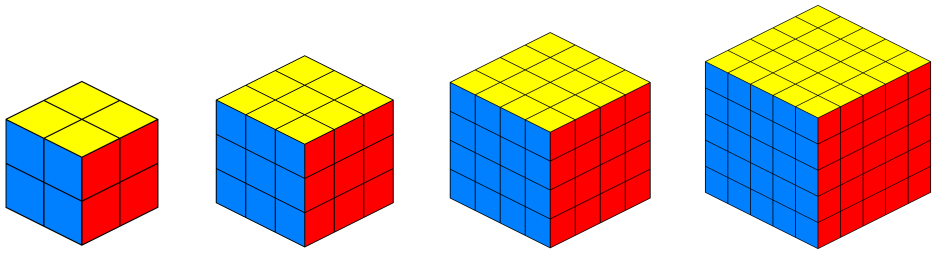


Figure 1.1: Cubes from 2x2x2 to 5x5x5

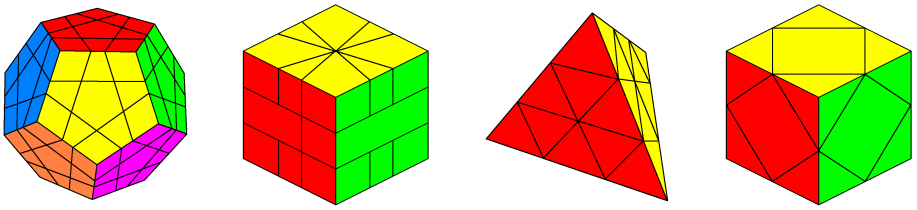


Figure 1.2: Megaminx, Square-1, Pyraminx, Skewb

Regular 3x3x3 solving however is the most popular event. Table 1.1 partially shows its ranking for official competitions. I'd like to emphasize that despite its competitiveness, the cubing community is a very open and friendly one. Of course everybody tries to be the best, but we do share solving methods, discuss improvements, and generally help each other to become better. Even at the competitions we have friendly races outside the actual competition, exchange ideas, and just spend time together having a lot of fun. This is just as important as the competition itself. It is also great to meet other cubers in person which you've only met online before because they live in other parts of the world.

There are also dozens if not hundreds of websites covering all kinds of topics around the cube and similar puzzles. There are tutorials, discussion forums, record lists, solving videos, blogs, etc.

1.2 Cube structure

Figure 1.3 shows a scrambled cube, one with one side solved, and one fully solved. Solving one side at a time is a popular method people try first, and often people say something like they once solved two sides. However,

Rank	Name	Seconds	Competition
1	Edouard Chambon	9.18	Murcia Open 2008
2	Ron van Bruchem	9.55	Netherlands 2007
3	Mitsuki Gunji	9.75	Kawasaki Open 2008
4	Erik Akkersdijk	9.77	Dutch Open 2007
5	Harris Chan	9.80	Toronto Open Fall 2007
6	Kouetsu Ando	9.83	Ibaraki 2007
7	Thibaut Jacquinot	9.86	Spanish Open 2007
8	Dan Dzoan	10.08	Caltech Spring 2007
⋮		⋮	
528	Yin Jia Qiu	19.95	UC Berkeley 2006
529	Dniel Fodor	20.00	World Championship 2007
⋮		⋮	
1152	Kaho Idekawa	29.96	Kawasaki Open 2008
1153	Patrycja Tucholska	30.00	Gdansk Open 2008
⋮		⋮	
2032	Dan Wilets	59.52	Rutgers Spring 2007
2034	Sung Dae-Kyu	1:00.00	KCA Korea Open 2008

Table 1.1: Partial 3x3x3 records list (as of March 27, 2008)

this approach is counterproductive. That's because we're not moving individual stickers around, but instead whole pieces holding several stickers together.

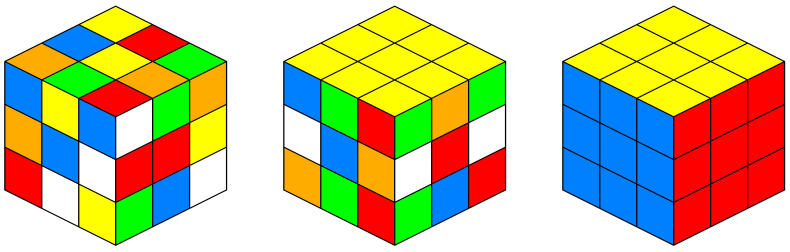


Figure 1.3: Cube scrambled, one side solved, solved

There are three types of pieces, shown in figure 1.4. The six center pieces each have one color sticker and never change position relative to each other. Because of this, they're usually thought of as reference, the yellow center standing for the yellow side, red center for the red side, etc. The eight corner pieces have three stickers, while the twelve edge pieces have two stickers. The center/corner/edge pieces are usually just called centers, corners and edges.

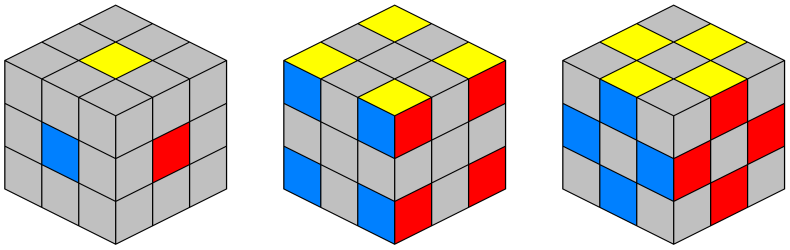


Figure 1.4: Cube centers, corners, edges

Have a look at the cube with one solved side again, and compare it to a cube with one whole layer solved (see figure 1.5). The yellow stickers appear to be where they belong, on the yellow side defined by the yellow center. But you can see that the other stickers around the top layer, on the same pieces as the yellow stickers, don't match on the left cube like they do on the right. So while the yellow stickers are apparently at the correct places, the pieces on which they reside are not, and thus even the yellow stickers are actually out of place. One should therefore consider not individual stickers but rather whole pieces, and put those at the correct places.

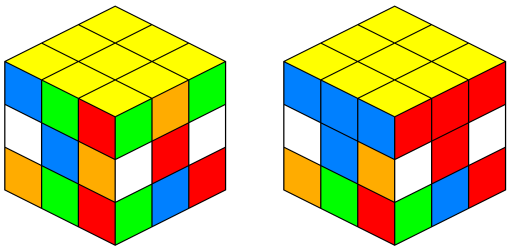


Figure 1.5: Solved side, solved layer

1.3 Algorithms and notation

Sequences of turns are called "algorithms". When you start solving a cube, there's nothing solved yet, and thus you can't break anything and have complete freedom of movement. Later, almost every single turn would break something. So instead of single turns, algorithms are used which do break something temporarily but restore it at the end, and which have a positive effect on the previously unsolved part.

When not shown in person by turning a real cube, algorithms are usually communicated using textual notation. The cube is held with both hands so that it has left, right, up, down, front and back sides. Notice the diagrams

used in this thesis rotate the cube a little for better visibility. They show the up, front and right sides.

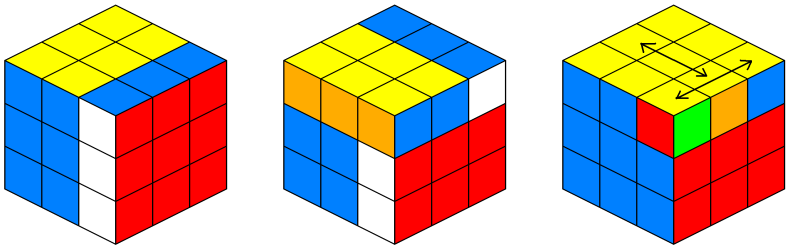


Figure 1.6: Effects of R, RU', and the T-permutation

The basic notation uses the six letters L, R, U, D, F and B to refer to the sides, with L=left, etc. The letter alone means a clockwise quarter turn (90 degrees) of the layer. Half turns (180 degrees) are denoted with a '2' (e.g., R2) and counterclockwise quarter turns with a single quote (e.g., R'). Figure 1.6 shows an initially solved cube after one R turn, then after an additional U' turn. An algorithm is simply written as its single turns in a row, so the second cube is turned by R U'.

The third cube shows the effect of the algorithm R U R' U' R' F R2 U' R' U' R U R' F'. The effect is called "T-permutation" because it swaps two corners and two edges in a T-shape (see arrows). These two swaps are the whole effect of the algorithm, no other pieces are affected[2]. It is thus very useful in the last steps of a solving method. The algorithm is also very finger-friendly and some people can apply its 14 turns in about one second.

There are different metrics to measure the "length" of an algorithm. They differ in what they allow to be counted as "one turn". For the 3x3x3 cube, the prominent three are:

- Only quarter turns of outer layers count as one turn.
- Any outer layer turn counts as one turn.
- Turning a middle layer counts as one turn, too.

With these metrics, a half turn of a middle layer would count as four, two and one turns, respectively. Other puzzles have analogous metrics. Which metric is used depends on the purpose. Counting outer layer turns feels the most natural and is also a good indicator of execution speed. Counting quarter turns is a bit strange, but often a better indicator for speed, and

[2] Actually the top center is rotated, but that's not visible on a normal cube.

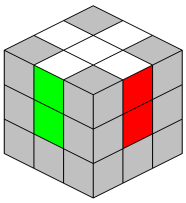
it offers interesting mathematical properties. Counting middle layer turns as one turn is valuable for methods making significant use of them. This thesis focuses on and uses the outer layer turns metric, as it is most popular.

1.4 Solving methods

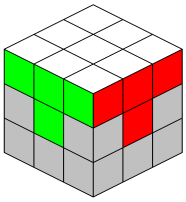
Many people have invented methods for solving the cube, the first being its inventor Ernö Rubik. The methods differ vastly, some solving the top layer first, others solving all corners first, yet others solving a 2x2x2 subcube first, and they differ in their later steps as well. This section describes a few methods and method categories.

1.4.1 Beginner method

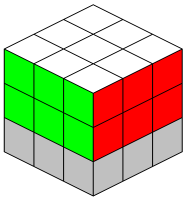
The typical beginner method works as follows:



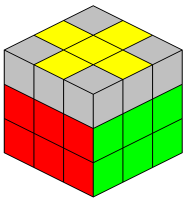
Start by solving four edges to build a cross on the top layer. Take care to have the outer stickers match their centers. Usually beginners solve one edge at a time.



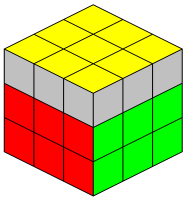
Add the four corners to complete the top layer, one piece at a time.



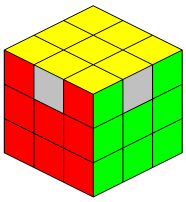
Add the four edges of the middle layer to complete the top two layers, one piece at a time.



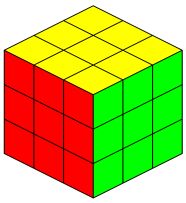
Turn the cube upside down for a better view, and flip the edges to build a top cross. Note that gray means we don't yet care about the outer stickers, only about the cross stickers. This step is called orienting the edges.



Rotate the corners to complete the top side. This is called orienting the corners. We're now finished orienting the last layer.



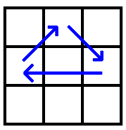
Move the corners to their correct position, keeping their orientation intact.



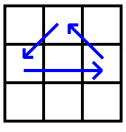
Move the edges to their correct position, keeping their orientation intact.

The first layer is solved one piece at a time, each piece with a few easy to understand turns. Most people are able to get this far on their own without help if they just try to, once they realize or are told they should solve the full layer, not just the side. The four edges of the second layer are solved one at a time, each with about 8 turns. This already is considerably harder than the first layer, as it becomes increasingly more difficult to solve more pieces while keeping the previously solved ones solved.

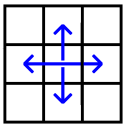
The last layer is the hardest, and few people solve it without help. Most simply learn algorithms for all possible cases, 16 for this method: three for edge orientation, seven for corner orientation, two for corner permutation and four for edge permutation. Below are the edge permutation cases with example algorithms solving them.



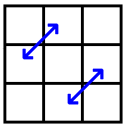
Clockwise 3-cycle: R' U R' U' R' U' R' U R U R2



Counterclockwise 3-cycle: R2 U' R' U' R U R U R U' R



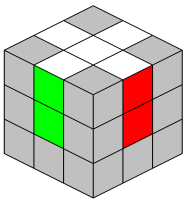
Swap two opposite pairs: F B U2 B' F' L' R' U2 R L



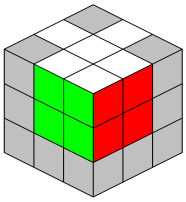
Swap two adjacent pairs: R B' R' B F R' B' F R' B R F2 U

1.4.2 Expert method

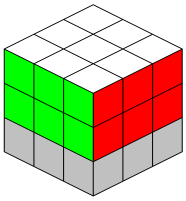
The currently most popular and successful expert method is called "CFOP", an abbreviation of its four steps Cross, First two layers, Orient last layer, and Permute last layer. It's also commonly called "Fridrich"[1], who had invented the method in the early 1980s and published the method on the internet in 1997.



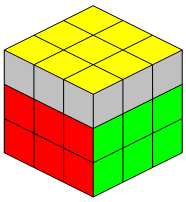
Start by solving four edges to build a cross. Unlike beginners, experts solve all four edges at once, in usually 5-7 turns. This is done rather intuitively with a few general principles.



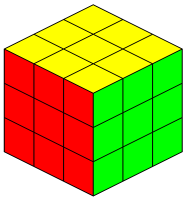
Notice how the cross solved the top two layers except four corner-edge pairs. Now one of those pairs gets solved. That is, corner and edge are solved simultaneously.



Solve the other three pairs to complete the first two layers. The order of the four pairs doesn't matter, so people solve the pair that they see first or that is easiest. Often both criteria point to the same pair.



Again let us rotate the cube upside down now, and orient the last layer in one go.



Permute the last layer in one go.

The above diagrams showed the cross and first two layers being built on the top, but that's only because it makes the explanation easier. In practice, they're usually built on the bottom. With enough practice, it's not necessary anymore to look at the pieces being solved at the moment. It's possible to instead look ahead, at the parts of the cube yet to be solved. The best solvers turn almost without interruption, recognizing the next step while solving the current one. So it's unnecessary and useless to see the already solved pieces and to not see the unsolved pieces. Besides this recognition advantage, solving with the cross on the bottom also allows faster twisting. The layer containing the cross rarely gets turned once the cross is built, but the opposite layer is free and gets turned a lot. And it's much faster to turn the top layer (simply pulling it with the index fingers) than the bottom layer, suggesting to put the often-turned layer on top and the rarely-turned layer on the bottom.

The cross is solved more or less intuitively, because there are far too many cases to learn solutions for them all. The remaining six steps however are solved with learned and practiced algorithms, so the cuber only needs to recognize the case and apply the algorithm learned for it. This is possible because the number of cases is rather small. Learning 41 algorithms for corner-edge pairs, 57 for last layer orientation, and 21 for last layer permutation is enough to be able to solve each of the seven steps in one go.

1.4.3 Other methods

The previously shown beginner method and the CFOP method fall in the ”layer by layer” category of methods. There are also very different methods, most falling in the general categories ”corners first” or ”block”.

Corners first methods usually have three solving phases: corners, some edges, remaining edges. Figure 1.7 outline a pure corners first method, solving only corners at first. Then the left and right layer are solved except for the top left and top right edge. The last phase solves the remaining six edges as well as the middle layer centers. This last phase can be done exclusively turning the top and the middle layer, which can be turned quickly.

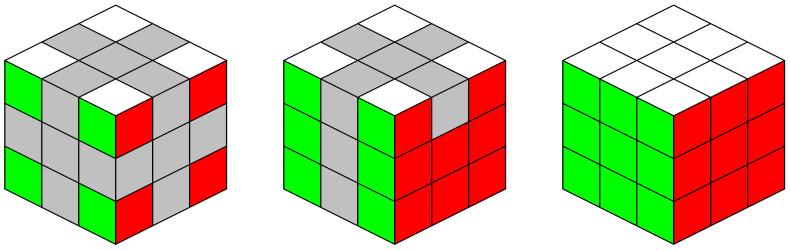


Figure 1.7: Typical corners-first stages

Today, very few people use corners first methods. In the early 1980s however they were dominant, with many of the competitors of the 1982 world championship using variations of corners first methods, including its winner Minh Thai.

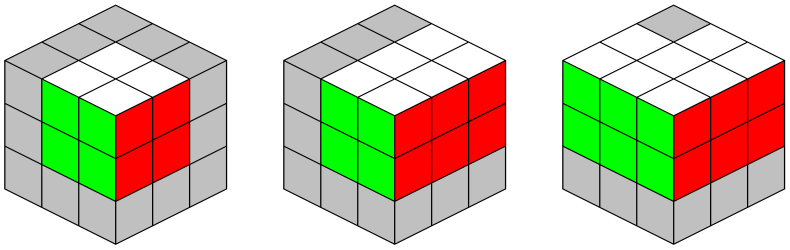


Figure 1.8: Typical block method

Block methods proceed by solving compact blocks. Usually they start by solving a 2x2x2 subcube of the whole cube as shown above, then extend it to a 2x2x3 block. After this they diverge, one possible next step being to solve another block so that two full layers are solved except one corner-edge pair. Finally they might solve that last pair and then the last layer like the CFOP method, though there are also other ways.

Compared to layer-by-layer methods, block methods usually require fewer turns. That's because they have a higher freedom of movement, i.e., more turns are possible without breaking what's already solved. After solving the cross for the CFOP method, only the side opposite the cross can be turned without breaking the cross, the other five sides break it. If instead you solve a 2x2x2 subcube first, then you can turn three sides without breaking that subcube. And even after the extending it to a 2x2x3 block, you can still turn two sides without breaking that block.

1.5 Common aspects of human solving methods

The methods shown in the previous section are just examples. Also, there are more methods and some don't even fall into the three mentioned general categories (layer by layer, corners first, blocks). Despite their differences, they however all share some common aspects.

Solve in several well-defined steps, each solving a certain few pieces. In some instances, pieces are only partially solved, usually oriented first and permuted later.

The first step solves about four pieces and is done rather intuitively, using general principles rather than learned algorithms.

Subsequent steps solve about two or three pieces at a time, using learned and practiced algorithms. Solving two or three pieces at a time seems to be optimal. Solving only one piece at a time is good for beginners, but it leads to an unnecessarily large number of turns. Solving more than two or three pieces at a time would create a large number of cases, which makes recognition hard and would require learning too many algorithms. The exception are the very last steps, when the number of cases becomes small enough.

When having a choice, for example which corner-edge pair to solve first in the CFOP method, people choose what they see first or what is easiest or fastest, which is often the same.

At this point I'd also like to mention that while most people simply learn and use a lot of algorithms they don't understand, this is not necessary. The cube is solvable without learning any explicit algorithms, but instead using some general principles. This way it is even possible to solve the cube and explain the exact purpose of every single turn. However, in many cases algorithms that aren't understandable are faster to execute, which is why people prefer them.

Chapter 2

Previous programs and results

Several computer programs have been developed to solve the cube, for various purposes.

- **Find shortest algorithms to solve a fully scrambled cube.** This is mostly interesting from a theoretical standpoint, as these solutions look rather magical to humans. The first turns don't appear to make sense, then more and more blocks are built and moved all around the cube, then all of a sudden everything falls into place in the last few turns.
- **Find shortest algorithms for specific effects**, e.g., the T-permutation. These algorithms can then be used inside human methods.
- **Determine how many turns are sufficient to solve any cube,** i.e., the badness of the worst cases. The exact number is not known yet, but it is somewhere from 20 to 25. There are cube states that were solved optimally and required 20 turns, providing the current known lower bound. The current known upper bound of 25 turns was proven by complicated extensive computer searches. This topic is useless for human solving, but extremely interesting from the computer science and theoretical viewpoints.
- **Generate scramble algorithms.** A puzzle can be scrambled somewhat randomly by a human, but computer-generated scrambles are of higher quality because humans tend to unconsciously repeat movements, causing non-random patterns. Current standard is 25 random non-canceling turns for the regular cube, more for larger puzzles. These still don't achieve a uniform distribution over the set of all cube states. An improvement currently discussed is to instead produce a

random cube state and then compute an algorithm to generate that state.

- **Find algorithms for pretty patterns** like a 2x2x2 cube rotated inside the 3x3x3 cube.

This chapter discusses some notable solver programs as well as lower and upper bounds for the number of turns sufficient to solve any cube.

2.1 Some numbers

The cube has about $4.3 \cdot 10^{19}$ possible states, and solving it optimally is still a hard problem for today's computers. The states can't be all covered in memory, as that would require almost five million terabytes even with just one bit for each state.

This can be reduced by symmetries. Applying the same scramble from two different angles results in two cubes that look different but are essentially equivalent. There are 24 angles: a choice of six sides as top side, then a choice of four sides as front side. Furthermore, mirroring a scramble also results in an essentially equivalent scramble. Together, this results in 48-fold symmetry. There are then about $9.0 \cdot 10^{17}$ non-equivalent states which would still require 100,000 terabytes. This is still too much for an exhaustive search, but some programs do exploit symmetries to reduce runtime and more importantly the size of lookup tables.

Already in 1981 some counting arguments were used to prove that there are cube states that require at least 18 turns to solve. They overestimate the number of states reachable from the solved state with N turns, which showed that 17 turns reach fewer than the total number of possible cube states so that there must be states that require more to be reached/solved. They're nonconstructive, proving only the existence of such cubes, but not providing any explicitly. They were however a good start for the search of lower bounds and weren't beaten until 1995. See appendix A for detailed descriptions of the counting arguments.

2.2 Thistlethwaite

The early human methods required up to about 100 turns to solve the cube. In 1981, Morwen Thistlethwaite introduced a method reducing this number

to 52 with a new method very much unlike previous ones[23]. It solves the cube in four steps, applying one algorithm in each step.

However, these steps are not defined by subgoals of pieces to solve, but by the turns allowed in the step, allowing fewer and fewer turns. The first step still allows all turns. In the second step, U and D are restricted to half turns only (i.e., no quarter turns anymore). The third step additionally restricts F and B to half turns, and the fourth step restricts all six sides to half turns.

The job of each step is thus to bring the cube to a state which can be solved with the remaining steps. For the first step, a set of 2048 algorithms is needed to make the cube solvable in the later steps. Those need 1082565, 29400 and 663552 algorithms, respectively. While too much for a human to learn, this method is very suitable for computers.

Thistlethwaite himself didn't provide full sets of optimal algorithms so his steps needed at most 7, 13, 15, and 17 turns, adding up to 52 turns. Later all steps were computed optimally and it was determined that 45 turns (7+10+13+15) suffice to solve any cube with this method, and 31.3 turns is the average length[22].

More powerful computers nowadays allow more powerful programs, but Thistlethwaite's idea remains very influential. It is also relatively easy to implement, as evidenced in 2004 by the results of Tomas Rokicki's cube programming contest[19]. The task was to write a short program to solve the cube and the programs were judged by a score combining program length, solution length, and runtime. The top three programs all implemented the Thistlethwaite method.

2.3 Kociemba's Cube Explorer

In 1992, Herbert Kociemba wrote a significantly better solver called Cube Explorer[6]. Basically he combined Thistlethwaite's first two and last two steps into one step each, resulting in a two-phase solver. Also, he doesn't stop searching when the first solution is found, but continues. There can be several phase-1 algorithms of the same shortest length to bring the cube into a state solvable in the second phase. They likely result in phase-2 solutions of different lengths, so testing all shortest phase-1 solutions and their continuations in the second phase can find shorter overall solutions. Furthermore, sub-optimal phase-1 solutions result in even more middle states and can lead to even shorter overall solutions if their phase-2 continuations happen to be short enough.

One could let the search continue until phase-1 solutions match the length of the shortest overall solution, at which point the solution is optimal. But this takes longer than optimal solvers which employ different concepts. Cube Explorer's real strength is quickly finding near-optimal solutions. It lets the user specify at what solution length the search shall be stopped, and it usually finds solutions with 20 turns in less than a second on a modern PC. Since most cube states require 18 turns to solve, this is very close. Kociemba himself once used it to solve a million random cubes and all of them were solved in 20 turns or less. He is also still improving Cube Explorer, adding functionality and adapting it to take advantage of the ever-growing power of computers.

Analyzing the two phases lead to a new upper bound of 29 turns to solve any cube. It was determined that the first phase needs at most 12 turns[25] and the second at most 18 turns[16], so a naive combination of both worst cases already provided an upper bound of 30 turns. It was then proven that the phase-2 cases requiring 18 turns can be avoided[18]. The restriction of phase 2 is that the R, L, F and B sides must not use quarter turns anymore, so the last turn of phase 1 must be one of those turns. However, that last phase-1 quarter turn done in the opposite direction would also work, i.e., would bring the cube to phase 2. It was checked that no two states requiring 18 turns in phase 2 were an R2, L2, F2 or B2 apart, meaning those cases can always be avoided by choosing the better direction for the last phase-1 turn.

2.4 New lower bound: 20 turns

In 1995 Michael Reid provided the first new lower bound since 1981's counting arguments[17]. He showed that the so-called superflip, where the cube is solved except all edges are flipped in place, requires 20 turns to solve. This state had already been conjectured to be hard, and a solution with 20 turns had been found in 1992 by Dik Winter[26]. To prove it optimal, Reid did an exhaustive computer search involving Kociemba's algorithm, though reduced manually considering the symmetries of the superflip as well as a characteristic of Kociemba's algorithm.

The reason it was thought to be hard and a candidate for a new lower bound was that no short solution had been found and that its high symmetry makes it a local maximum, i.e., no neighbor states require more turns to solve. Because of the symmetry it doesn't matter whether you apply an R turn or an F' turn or any other quarter turn to the superflip, they all lead to equivalent states (similar to making any quarter turn to the solved state,

which might be easier to visualize). The same applies to half turns. So basically the superflip has just two different neighbors, let's say those an R turn and an R2 turn away. One of them must be closer to solved (only the solved state itself has no neighbors closer to solved) and the other can't be farther from solved because the two neighbors are neighbors of each other as well, so their distance to solved can't differ by more than one turn.

2.5 Korf's pattern databases

In 1997, Richard Korf[9] wrote the first program to solve random cubes optimally. He solved ten random cubes and all were solved in 18 or fewer turns. He used IDA* with three large heuristic functions stored as lookup tables in memory, called *pattern databases*. One heuristic told the minimum number of turns required to solve just the corners of the cube, another covered six specific edges, the last covered the other six edges. So given an arbitrary cube, the program could quickly look up how many turns these parts of the cube would require, which provided a lower bound for the whole cube.

These lookup tables together took 82 megabytes of memory. One table for seven edges would've required 244 megabytes and presumably back then this much memory was not easily available as Korf didn't do this despite it likely improving the performance considerably. In the same paper, he also carried out a performance analysis of his cube findings, suggesting a linear relationship between the amount of memory used and the speed IDA*.

Several people later implemented solvers based on the same concept as Korf. The Hume program I developed for this thesis is somewhat one of them, although it uses the concept in a different way for a different purpose. I do however intend to write my own optimal specialized 3x3x3 solver later.

2.6 Jelinek's ACube

In 2000, Josef Jelinek published the first version of yet another solver, which he called ACube[5]. It basically uses the same algorithm as Kociemba's Cube Explorer, but it allows to ignore parts of the cube, either the whole piece or just its position or orientation. This is extremely useful for finding algorithms to solve the early steps of human methods, since at that point you have a few pieces solved, some are about to be solved, and you don't care about the rest which will be looked at and solved later. ACube let's you

declare this uncared-for rest as such. The result is that you get an algorithm which keeps solved what you want, solves what you want, and usually mixes up the rest which you don't care about.

Without this functionality, you'd have to choose between two evils. If you just want any algorithm, you could tell the program to also keep the rest as it is. However, that usually results in longer algorithms than if that rest is allowed to get mixed up. If you want to get the shortest algorithm, you could try all possible mixups of the rest. But that would usually be a lot of cases. This is why ACube is extremely useful in practice, since except for the very last method step, there's always a rest you don't care about, and ACube lets you find optimal algorithms for exactly that. Other advantages of ACube are the ability to show all optimal (as well as many near-optimal) algorithms instead of just one, and an option to optimize for quarter turns, which often leads to faster, more finger-friendly algorithms.

2.7 Norskog's 4x4x4 solver/analysis

In 2006, Bruce Norskog created a program for solving the 4x4x4 cube[11]. This puzzle is far too complex to be solved optimally. Remember the 3x3x3 cube has about $4 \cdot 10^{19}$ possible states. The 4x4x4 cube has about $7 \cdot 10^{45}$ possible states. Norskog's solution is a five-phase solver, each phase is solved optimally but the overall solution is sub-optimal (except in pathological extreme cases). His main objective though was not to have a solver, but to analyze the solution and provide an upper bound for the number of turns required to solve the 4x4x4 cube. For this he used an exhaustive computer search building complete tables of optimal solutions for all five phases, and determining the worst cases for all of them. These are 11, 18, 14, 17 and 19 turns, resulting in an overall worst case of 79 turns. He later improved that to 77 turns and analyzed different metrics to count the turns (you can for example allow or forbid pure inner slice turns), resulting in worst cases of 82 and 67 turns[12].

It is interesting to note that the basic idea is the same as Thistlethwaite's. Norskog's first phase allows all turns, the fifth phase allows only half turns and no quarter turns, and the phases in between allow fewer and fewer turns. Yet another example of Thistlethwaite's idea still remaining useful and influential today.

2.8 My short solver

In Tomas Rokicki's cube programming contest[19] mentioned earlier, I actually submitted two very different programs. One was the aforementioned Thistlethwaite solver, another was a Perl program that was by far the shortest program of all, easily fitting on one computer screen with lots of unused space. Despite being the shortest, it ended up in the middle of the ranking because the solutions it provided were very long (several hundred turns).

The main idea was to not think in terms of single turns, but instead in terms of algorithms with specific small effects, each about solving one piece. This single-handedly removed the need for a real cube simulator. The program can solve any cube, but it actually doesn't simulate a single turn. It didn't have an interesting internal data format to represent the cube, either. It simply worked on the human-readable representation it got as input.

Somewhat as a by-product, I later developed methods for solving the cube blindfolded based on this approach of solving one piece at a time. They're very easy and have become fairly popular and successful.

2.9 Closing the gap

As mentioned earlier, it is not known how many turns suffice to solve any cube, and the current lower and upper bounds are 20 and 25. This is an active field of research with sophisticated attempts to improve both bounds and close the gap. Silviu Radu reduced the upper bound to 28 in 2005[15] and to 27 in 2006[14]. Daniel Kunkle and Gene Cooperman reduced it to 26 in 2007[10]. Tomas Rokicki reduced it to 25 in 2008[21]. Kociemba, Rokicki and some others are discussing plans to reduce it to 24 and possibly even 23 in the near future[8].

At the opposite end, the lower bound of 20 turns hasn't been improved since 1995. But that doesn't mean it hasn't been attempted. When Kociemba solved a million random cubes using his near-optimal Cube Explorer, he did so to find some cubes that the program couldn't solve in 20 turns or less[7]. These would've been candidates for requiring more than 20 turns and would've been tested with an optimal solver. But he didn't find any. Cube Explorer solved all one million cubes in 20 turns or less. In 2006, Tomas Rokicki searched months for cubes that require 20 or more turns, using specialized solvers that test huge numbers of cubes in parallel[20]. He found about half a million that need exactly 20 turns, but none that

need more. Overall he tested $7.6 \cdot 10^{14}$ cubes, about 1/53000 of all possible cubes. Later in 2006, Silviu Radu analyzed all $1.6 \cdot 10^{11}$ cubes that possess any symmetry[13], because hard cases found earlier had been symmetric, particularly the superflip. He found about a million cubes that require 20 turns, but again, none required more.

While we don't yet know what the end result will be, many if not most people believe it to be 20 turns, and probably nobody would be surprised if it were.

Chapter 3

Hume

This chapter discusses briefly why previous programs are unfit for evaluating whole solving methods. Then it introduces Hume, the "human method evaluator" program, developed as part of this thesis. Its purpose is precisely the evaluation of human methods, and it is flexible enough to be used for puzzles other than the standard Rubik's cube. This chapter focuses on the point of view of the user, chapter 4 uses Hume to analyze and compare some human methods, and chapter 5 talks about Hume's implementation.

3.1 Shortcomings of previous programs

There already are programs for solving several puzzles, though mostly just for the standard 3x3x3 cube. Some of them have been described in the previous chapter. They can be used to find algorithms for human methods, but this already requires a lot of work: determining all cases that can occur in the method steps, entering them into the solver programs, and letting the programs search for algorithms. Furthermore, even after algorithms are found for all cases, it's still a lot of work to combine all the data into an evaluation of the method as a whole.

This gets even harder because the algorithms usually don't tell the whole story. Have a look at the left two cubes in figure 3.1. The cross for the CFOP method has been solved on the bottom and now the white/red/green pair shall get solved next. The two cubes differ, but they basically show the same case, just a quarter turn of the top layer apart. And there are two more equivalent cases reached by turning the top layer one or two quarter turns further. Usually people learn just one algorithm, not four, and adjust the top layer to the known position before applying the known algorithm.

In general, people often do this kind of setup turn between applying the learned algorithms. Another issue is shown in the third cube: The corner is right above the slot where it belongs, but its corresponding edge is hidden in a wrong slot. Learning direct solutions for all these cases would require learning a lot more algorithms, and it's unnecessary. Instead one would first pull the edge out of the wrong slot so that both pieces are in the top layer, and then (possibly after another setup turn) apply an algorithm to solve the pair from there. In summary: People don't actually find and learn algorithms for really all cases, but only for a reasonable number of nice cases, and then use extra turns between the algorithms.

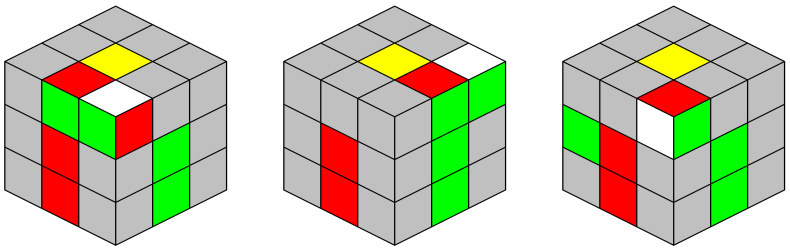


Figure 3.1: Case equivalence, hidden pieces

So determining cases and finding algorithms for them is not only a lot of work, but it's also not even sufficient to evaluate the method as a whole. Thus the usual solver programs might be good at what they're doing, but they're hardly helpful in quickly evaluating a method. It's simply not what they were intended for.

Which is exactly what the Hume program attempts. The main goal is to let the program do all the dirty work, requiring a minimum of work for the user to describe the method. This is where the observation about how human methods work comes into play. The user ought not have to enter a lot of cases for the method steps, but instead only enter a high-level description of the method itself, by telling the program about the goals of the method steps.

3.2 The Hume program

The observation of commonalities between human solving methods as well as a desire to compare them in an easy and automated way was the motivation for the program developed for this diploma thesis. Its name is Hume, an acronym for human method evaluator. The idea is to describe a puzzle and a human solving method to the program, which shall then evaluate the method by randomly scrambling and solving the puzzle using the

given method and collecting statistics about the solves. Doing this for several methods then allows a comparison of them, and it also offers a quick evaluation of a new method that someone invented without having to do a lot of tedious work.

Another design goal of Hume is to not specialize in the standard 3x3x3 cube like most programs do, but to be flexible enough to cover other puzzles as well. This is achieved by letting the user describe the puzzle.

3.3 Execution

The program is a Java command line application. It takes input from the standard input stream and prints output to the standard output stream. For ease of use, it's recommended to use files for input and output. Example:

```
java Hume <in.txt >out.html
```

3.4 Input

The input describes the puzzle, the solving method, and tasks telling the program what to do. Example:

```
puzzle
U = [YO YG YR YB] + [YOG YGR YRB YBO]
D = [WO WB WR WG] + [WOB WBR WRG WGO]
L = [GY GO GW GR] + [GYO GOW GWR GRY]
R = [BY BR BW BO] + [BOY BYR BRW BWO]
F = [OY OB OW OG] + [OYB OBW OWG OGY]
B = [RY RG RW RB] + [RYG RGW RWB RBY]

method
WO WB WR WG
WOB OB | WGO GO | WRG RG | WBR BR

tasks
solve 100 scrambles of 50 turns
```

This describes the standard Rubik's cube, the cross and four corner-edge pairs method, and tells the program to solve 100 random scrambles of 50

turns each. The three sections start with the keywords **puzzle**, **method** and **tasks** and will now be described in more detail.

3.4.1 Describing the puzzle

In the **puzzle** section, the puzzle is described by naming the turns and stickers and specifying the effects of the turns on the stickers. The sticker names follow a certain scheme, see figure 3.2. The name starts with the letter for the color of the sticker itself followed by those of the other stickers on the same piece in clockwise order. For example, the yellow sticker on the yellow/orange/green corner is called YOG. The turns describe the cyclic changes of the stickers, for example the U turn causes the cycle of edges [YO YG YR YB]. The notation means that the sticker on spot YO moves to the YG spot, YG to YR, YR to YB, and finally YB to YO. This is also visualized in figure 3.2.

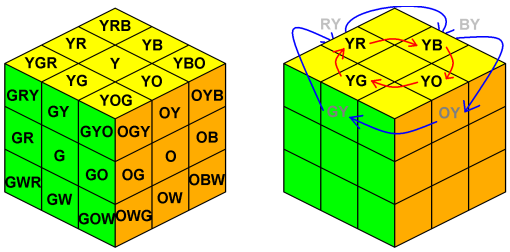


Figure 3.2: Sticker names, edge effects of turning U

The edge stickers on the outside of the layer are implicitly included as well so it's unnecessary to add [OY GY RY BY]. The effect on the corners is described equivalently, for U it is [YOG YGR YRB YBO]. Again it's unnecessary to add the two cycles [OGY GRY RBY BOY] and [GYO RYG BYR OYB] explicitly.

3.4.2 Describing the solving method

The **method** section describes the solving method by specifying its subgoals. In the example this is done by these two lines:

```
WO WB WR WG
WOB OB | WGO GO | WRG RG | WBR BR
```

Subgoals that can be done in any order are put on one line, separated by pipes. To specify a certain order, the subgoals are put on separate lines,

earlier lines being solved earlier.

A subgoal is given as a set of stickers that shall be solved. In the example, the first subgoal is solving the stickers WO, WB, WR, and WG. This is the white cross. Next, on the second line, the program is told to solve four subgoals in any order, and each of these subgoals is one of the four corner-edge pairs needed to add to the initial cross to complete the first two layers.

Remember the stickers have unique names so that even two stickers of the same color are distinguished, for example WO differs from WB, the former meaning the sticker on the edge with the an orange sticker, the latter meaning the edge with the blue sticker. Because of this it is enough to specify just one of the stickers on a piece. Solving WO implies also solving OW.

3.4.3 Describing what to do

The **tasks** section finishes the input, telling the program what to do with the puzzle and the method. There's currently only one command:

solve **S** scrambles of **T** turns

This tells the program to scramble and solve the puzzle **S** times, using **T** turns for each scramble. The latter is necessary because the program has no idea how many turns scramble the puzzle well enough.

3.5 Solving and output

Once the program is started, it first reads and analyzes the input and prepares some auxiliary data from the puzzle and method descriptions. Then it is ready to execute the tasks requested in the input. An HTML page like in figure 3.3 is produced, first showing overall statistics for the whole solves as well as for the method steps, which can be evaluated to judge the method and compare it to others, as demonstrated in chapter 4. The output also shows the individual solves so that the user can study them in more detail and hopefully learn something from them[1].

To solve the puzzle, Hume applies the given method by solving its subgoals in the given order. In case of parallel subgoals like the four corner-edge pairs of CFOP, it greedily chooses to solve the one with the shortest solution first. This is done to reflect how humans solve, as observed in chapter 1.

[1] Lars Vandenbergh's ImageCube PHP script[24] is used to display the solves visually.

solution length	Frequency
16	1
19	3
20	8
21	6
22	16
23	14
24	20
25	13
26	13
27	6
Average = 23.38	

Subgoal #1	Frequency
2	1
4	8
5	48
6	42
7	1
Average = 5.33	

Subgoal #2	Frequency
3	3
4	13
5	29
6	43
7	12
Average = 5.48	

Subgoal #3	Frequency
3	1
4	9
5	22
6	31
7	29
8	7
9	1
Average = 6.03	

Subgoal #4	Frequency
3	4
4	10
5	6
6	12
7	40
8	27
9	1
Average = 6.59	

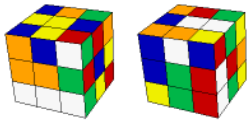
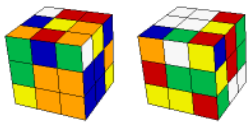
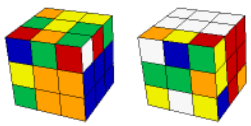
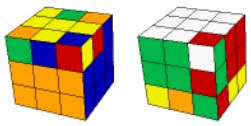
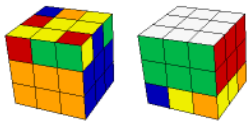
Solve #1		
	B1 L2 U3 L3 U2 R2 U3 F3 U2 U3 F2 R3 F3 F2 L1 U3 R1 U3 L1 U3 L2 F2 B1 D1 F2 L1 R2 L1 L3 F2 L3 B2 R3 R3 B3 D1 L2 D3 B1 F2 D1 D2 R2 B3 D3 L2 D3 B3 D3 U1	
5	F3 R3 B3 L3 F3	
6	U1 L3 B1 U2 L3 B1	
7	U1 B1 U1 L2 U3 B3 L1	
8	L1 U3 L3 B3 R3 U2 R1 B1	
Total solution length: **26**		

Figure 3.3: Partial output of the Hume program

Hume solves each subgoal in the smallest number of turns. In reality, humans often use algorithms that take slightly more turns but which can be executed faster because they allow a more finger-friendly flow of turns. So in this respect, Hume doesn't accurately represent human solving, but it gets close and is a good indicator. When comparing two methods with Hume, both are affected by this deviation, and I conjecture to a similar degree. Also, in fewest moves contests we do have a lot of time and try very hard to find shortest solutions counting the number of turns and don't care about execution speed, so for this category the approach is more accurate. For speedsolving, ultimately a more complex metric than counting the turns should be employed. Instead, the positions of the fingers as well as difficulty and speed of turns and turn combinations ought to estimate the "length" of an algorithm. To my knowledge, this has not been done yet by anyone, although it would be a very useful addition to the speedcuber's toolbox for finding better algorithms and methods.

3.6 Overlapping subgoals

Consider the block-style method for solving the top two layers starting by solving a 2x2x2 subcube and extending it to a 2x2x3 block. Depending on which of the four possible subcubes you begin with, different pieces provide an extension. Figure 3.4 shows the four possible starts, their respective extensions marked with dark gray.

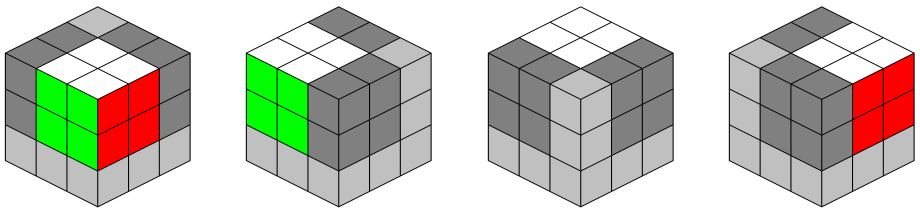


Figure 3.4: Different block-style starts and extensions

If you don't want to specify a certain subcube to begin with but want to allow all, how do you encode the second and later subgoals? They depend on the program's choice for the first subcube during runtime, so unconditionally specifying which pieces to solve next doesn't work.

In a future version of Hume I might extend the language for describing the method, one option being conditional specifications. But the above method and others can already be encoded by using overlapping subgoals. Rather than talking about the different block types to be added (2x2x2, 2x2x1, 1x1x2) at different times and places, this method can neatly be described

as building the four 2x2x2 blocks in figure 3.5. Solving two adjacent subcubes means solving the 2x2x3 block they make up.

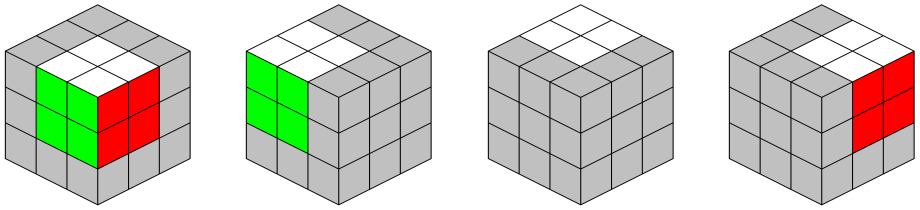


Figure 3.5: Four overlapping 2x2x2 subcubes

The blocks overlap, but that doesn't hurt, and it makes the description of the method much easier because we don't have to describe the different block sizes and dependencies between the blocks. Also, it allows the second block to be diagonally opposite of the first one, in case that is the shortest extension. While that is admittedly not intended by the original method, it is worthwhile to find out whether this would be beneficial. Once I checked 100 solves and Hume extended the first subcube diagonally only five times. If this is still not acceptable, another technique is to combine the statistics of a fixed and a free start appropriately. This is shown in the next chapter.

Overlapping subgoals can also be used to speed up the search a lot. Consider the input for the cross and four pairs again:

```
WO WB WR WG
WOB OB | WGO GO | WRG RG | WBR BR
```

Hume uses the subgoals to create heuristics for the search. The larger the subgoals, the better the heuristics. For that reason it is beneficial to encode the method as follows:

```
WO WB WR WG
WOB OB WO WB | WGO GO WG WO | WRG RG WR WG | WBR BR WB WR
```

This is the same as before, except each pair subgoal now also includes the two already solved cross edges next to it. Theoretically, this is an equivalent description, but in practice this makes the search much faster. Pitting the two versions against each other in 20 sample solves, the pure pair version on average took 35 times as long to solve and visited 18 times as many search states.

Chapter 4

Evaluation of Rubik's Cube methods

This chapter analyzes and compares two human methods for solving the cube, using Hume. However, only the first two layers of the cube will be covered here, not the last layer, for these reasons:

- It has already been done. There are relatively few cases for the last layer and all of them have already been solved optimally[2] in 2001. This includes algorithms and statistics for solving the last layer with the usual and some unusual methods for solving the last layer.
- The last layer is significantly harder to compute. Most of the cube is already solved, there's thus less freedom of movement, more pieces need to be tracked, and more turns are needed in each step. More turns means deeper search which requires a lot more time and memory. There are programs that are able to do this, but even they can take a long time for hard cases. Also, they're optimized for the 3x3x3 cube, for example using highly efficient pruning tables specific to that puzzle. Hume on the other hand is designed to be flexible and allow the user to describe not just the 3x3x3 but also other puzzles, which comes with a performance cost.
- Solving the last layer is rather boring. While the first two layers allow a lot of freedom and creativity, the last layer is all about recognizing the current case and blindly applying an algorithm for it.

You might notice a rather personal style in this chapter. That is because I relate the findings gained with Hume to my personal experience and knowledge in speedcubing, and I hope you won't mind.

4.1 Analyzed methods

The two analyzed methods are CFOP and the block-style method described in chapter 1 in more detail. As only the first two layers (F2L) are covered here, I will refer to them as CrossF2L and BlockF2L. CrossF2L has five steps, solving the cross first and then adding four corner-edge pairs. I specified the method as a cross followed by four overlapping subcubes as discussed at the end of the previous chapter.

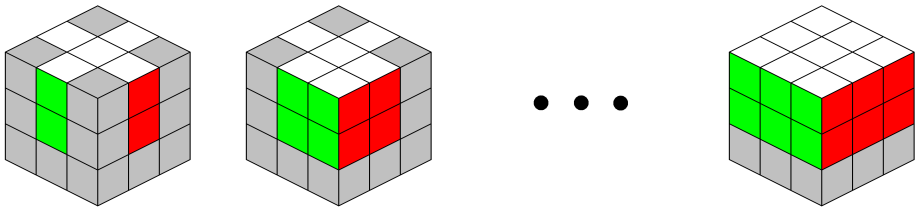


Figure 4.1: CrossF2L: Cross, then corner-edge pairs

BlockF2L uses four steps. It starts by solving one 2x2x2 subcubes and gradually extends it with more blocks to the full first two layers. I specified the method as four overlapping subcubes as discussed at the end of the previous chapter.

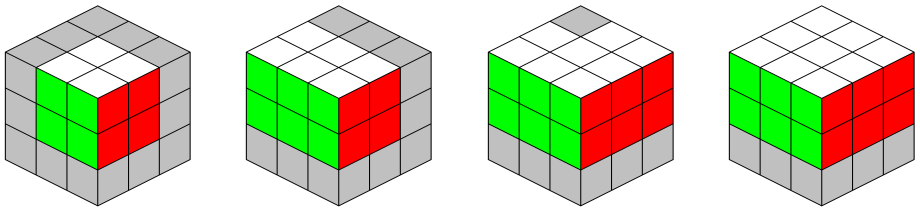


Figure 4.2: BlockF2L: Subcube, extended by more blocks

For CrossF2L I chose to always solve the white cross, and similarly BlockF2L always includes the white side. It is of course possible to instead start with any of the other five colors, and some people don't always start with the same color but the one they find easiest, deciding anew for each cube they get. Most cubers however, including the fastest, always start with the same color. This makes recognition easier because they always have to look out for the same sets of pieces. Among block-style cubers there seem to be more who are color-neutral than among the cross-style cubers, but I don't know to what extent. I believe fixing which two layers to solve, but not fixing in what order to solve its four subcubes, is a good compromise.

4.2 Results

I used Hume to solve the first two layers of 10000 randomly scrambled cubes with CrossF2L and BlockF2L. The overall comparison is shown in table 4.1 and figure 4.3. On average, BlockF2L beats CrossF2L by 5.1 turns. This is not surprising, as BlockF2L has one less step and its compactness offers more freedom for movement without breaking already solved parts. However, the shorter length comes at a price.

	CrossF2L		BlockF2L	
Turns	Frequency	Cumulative	Frequency	Cumulative
13			0.01%	0.01%
14			0.02%	0.03%
15			0.02%	0.05%
16			0.12%	0.17%
17	0.01%	0.01%	0.25%	0.42%
18	0.02%	0.03%	0.84%	1.26%
19	0.08%	0.11%	1.88%	3.14%
20	0.14%	0.25%	3.72%	6.86%
21	0.21%	0.46%	6.88%	13.74%
22	0.54%	1.00%	11.40%	25.14%
23	1.11%	2.11%	16.60%	41.74%
24	2.09%	4.20%	19.96%	61.70%
25	4.22%	8.42%	18.93%	80.63%
26	7.15%	15.57%	12.48%	93.11%
27	10.27%	25.84%	5.27%	98.38%
28	14.50%	40.34%	1.40%	99.78%
29	17.84%	58.18%	0.20%	99.98%
30	16.73%	74.91%	0.02%	100.00%
31	13.31%	88.22%		
32	8.00%	96.22%		
33	2.99%	99.21%		
34	0.69%	99.90%		
35	0.07%	99.97%		
36	0.03%	100.00%		
Average	28.85		23.74	

Table 4.1: CrossF2L versus BlockF2L

Both methods start by solving four pieces at once, making their first step about equivalent. But afterwards BlockF2L solves three pieces at a time while CrossF2L solves only two at a time. Recognition for three pieces is harder and there are many more cases, too many to learn and practice algorithms for all of them. At this point it is useful to look not just at the

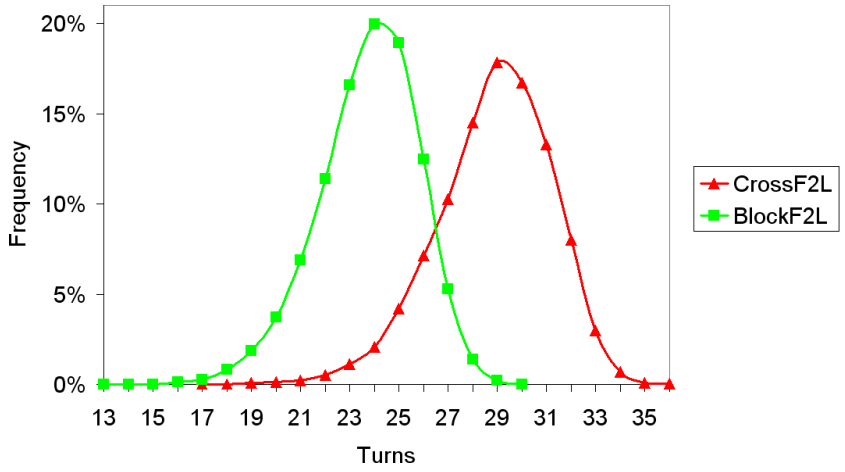


Figure 4.3: Graphical comparison between CrossF2L and BlockF2L

total number of turns, but at the number of turns the methods need for each step. These statistics provided by Hume are shown in the left part of table 4.2 and figures 4.4 and 4.5. Note the cross is left alone and the pairs are pitted against the blocks because they match rather well.

Greedy order			
CrossF2L		BlockF2L	
Cross	5.82		
Pair 1	5.03	Block 1	5.27
Pair 2	5.40	Block 2	5.49
Pair 3	5.80	Block 3	6.20
Pair 4	6.81	Block 4	6.79
Total	28.85	Total	23.74

Fixed order			
CrossF2L		BlockF2L	
Cross	5.83		
Pair 1	6.09	Block 1	6.08
Pair 2	6.09	Block 2	5.97
Pair 3	6.17	Block 3	6.68
Pair 4	6.72	Block 4	6.59
Total	30.90	Total	25.32

Table 4.2: CrossF2L/BlockF2L steps, greedy versus fixed order

First notice that the fourth pair and the fourth block both need about the same number of turns. This is expected, as the fourth block *is* a pair. Theoretically they ought to require exactly the same number, but then again, this is a randomized experiment and not an exact computation. Now let us look at the earlier steps. The first block averages 0.55 turns less than the cross, despite having the same number of pieces. Personally I find solving that block harder, it takes me much more time to think it through and also takes me more turns than solving the cross. But this has to do with practice. I'm a CFOP solver, so I have a lot of practice and little guidelines for solving the cross. Cubers who have practiced block-solving a lot have no problem with doing so. Similarly, solving the CrossF2L pairs has become second nature to me, it is now an almost subconscious repetition of

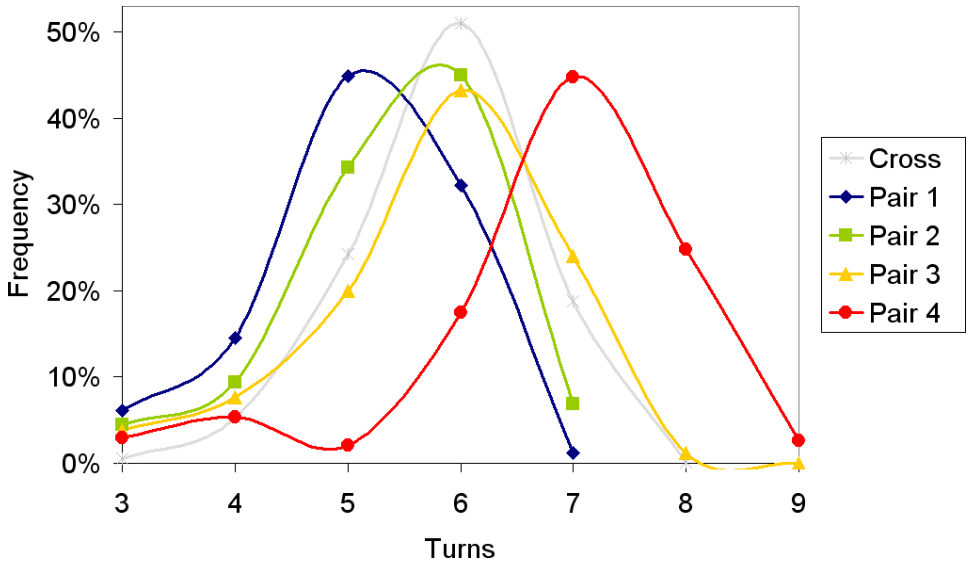


Figure 4.4: Graphical statistic for CrossF2L

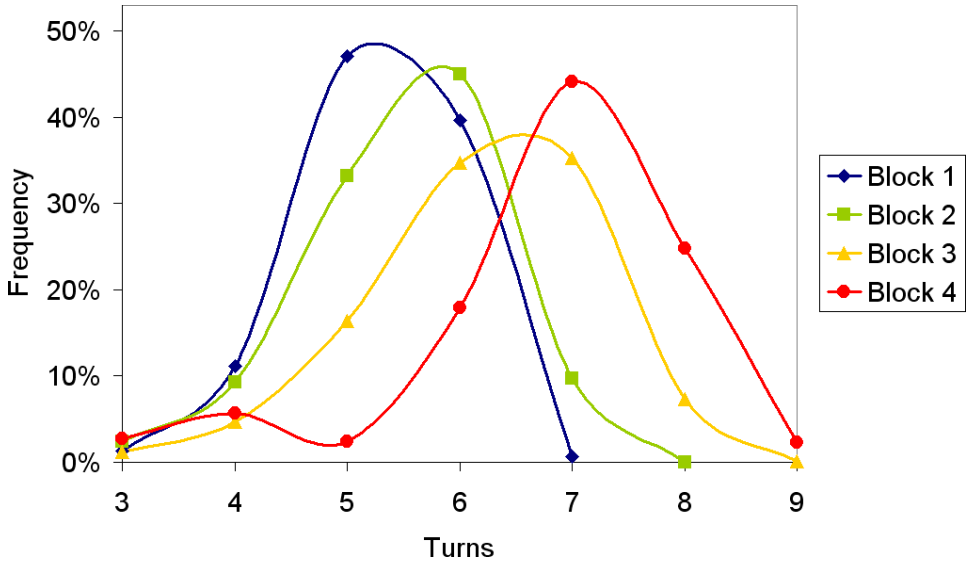


Figure 4.5: Graphical statistic BlockF2L

recognizing patterns and executing corresponding algorithms. And again I struggle with extending the BlockF2L block while block-solvers don't. So probably practice and knowing little guidelines for block-solving can make it fairly easy, too. The statistics also suggest that BlockF2L isn't very hard, as despite solving more pieces at a time, the number of turns required at each step is about the same as for CrossF2L.

The right part of table 4.2 shows the effect of specifying a fixed order for the pairs/blocks, in contrast to the free order that always solves the easiest choice first. This was determined with 1000 solves and the method specifications were modified so that the pairs/subcubes must always start with the same pair/subcube and then complete the remaining pairs/subcubes in clockwise order. For me very surprisingly, this increased the number of turns only by 2.05 for CrossF2L and even just 1.58 for BlockF2L. Also surprising at first is that the fourth pair/block becomes shorter, but that can be explained easily. One of the hardest cases for a pair is when the edge is already solved and the corner is at the correct place but rotated incorrectly. With the greedy easy-comes-first strategy such a case might be left to be solved last, while fixed order solving is likely forced to solve it earlier. I suspect the same phenomenon to be responsible for the even stronger effect in BlockF2L.

Finally, tables 4.3 and 4.4 provide as much detailed statistics for CrossF2L and BlockF2L as Hume delivers. It shows how frequently one can expect to solve each step in a certain number turns (or less, looking at the cumulative columns). Of particular interest are the worst cases and how rare they are or aren't. Personally I am baffled how often Hume solves CrossF2L pairs in five turns. All pairs that take three or four moves are easy to see, but I rarely solve pairs in five turns, mostly I take six to eight. Hume on the other hand solves the first pair in five turns or less in 2/3 of all cubes, and less but still quite often repeats the feat with the later pairs.

Another revelation is that the first pair of CrossF2L has a one percent chance to be solved completely accidentally while solving the cross, so that it requires zero turns. You might wonder why zero turns appears somewhat frequently but one and two turns don't. This is simply caused by the lack of freedom of movement in CrossF2L. If a pair is not solved already, it takes at least three turns to solve it. The cross must be temporarily broken, which takes one turn, and restored, which takes another. And something must be done between those two turns. Similarly, why was only the first pair ever solved accidentally? This is because if the cross happens to solve some pair, the greedy method will immediately grab that pair as the first to "solve". Also, solving a pair affects the other pair slots only a little, so solving a pair is unlikely to accidentally solve another.

	Cross		Pair 1		Pair 2		Pair 3		Pair 4	
T	F	C	F	C	F	C	F	C	F	C
0			1.0	1.0						
1	0.0	0.0								
2	0.1	0.1								
3	0.6	0.7	6.1	7.2	4.5	4.5	3.8	3.8	2.9	2.9
4	5.2	5.9	14.6	21.7	9.4	13.9	7.7	11.5	5.3	8.2
5	24.2	30.1	44.9	66.6	34.3	48.1	19.9	31.4	2.0	10.3
6	51.0	81.2	32.3	98.8	45.0	93.1	43.2	74.7	17.5	27.8
7	18.8	99.9	1.1	100.0	6.9	100.0	24.1	98.7	44.8	72.6
8	0.1	100.0					1.2	100.0	24.8	97.4
9							0.0	100.0	2.6	100.0
A	5.82		5.03		5.40		5.80		6.81	

Table 4.3: Detailed CrossF2L steps (T=turns, F=frequency, C=cumulative, A=average)

	Block 1		Block 2		Block 3		Block 4	
T	F	C	F	C	F	C	F	C
0								
1			0.04	0.04	0.03	0.03		
2	0.15	0.15	0.34	0.38	0.29	0.32		
3	1.29	1.44	2.37	2.75	1.16	1.48	2.68	2.68
4	11.11	12.55	9.26	12.01	4.75	6.23	5.71	8.39
5	47.07	59.62	33.22	45.23	16.34	22.57	2.43	10.82
6	39.67	99.29	45.03	90.26	34.72	57.29	17.91	28.73
7	0.71	100.00	9.71	99.97	35.31	92.60	44.19	72.92
8			0.03	100.00	7.33	99.93	24.79	97.71
9					0.07	100.00	2.29	100.00
A	5.27		5.49		6.20		6.79	

Table 4.4: Detailed BlockF2L steps (T=turns, F=frequency, C=cumulative, A=average)

4.3 Analysis of found solutions

Besides the statistics, I was also very interested in studying the individual solves that Hume found. Of course I didn't study the many thousands accumulated during the searches, but I picked some of average and shorter than average length. Note that for CrossF2L, "average" for Hume is already several turns better than what I myself average. And for BlockF2L I'm hopelessly lost.

The solutions for BlockF2L were often amazing, but as I don't have much experience with that solving style, I could not judge them properly. Closer looks at CrossF2L solutions however were often hilarious to me, actually making me laugh. For the past five years I have maybe solved the cube a few dozen times per day, so I have seen all cases for the pairs thousands of times. But what Hume came up with was new to me.

While I will most likely never attain the level of Hume, I am beginning to see small patterns and how to deal with them, something I called little guidelines earlier in this chapter. Studying more solves in more detail as well as practicing will hopefully enable me to find shorter solutions in less time, both for CrossF2L and for BlockF2L. I doubt it will make my speedcubing much faster, but it should improve my understanding and knowledge of the cube and be very useful in the fewest moves competitions.

4.4 Tripod and combining statistics

One idea for new methods that has come up in the cubing community is called "Tripod". Rather than solving via two full layers, it first solves all but seven pieces in a tripod shape. One way to reach that state is to start with a 2x2x2 subcube and extend it on its three sides, as shown in figure 4.6.

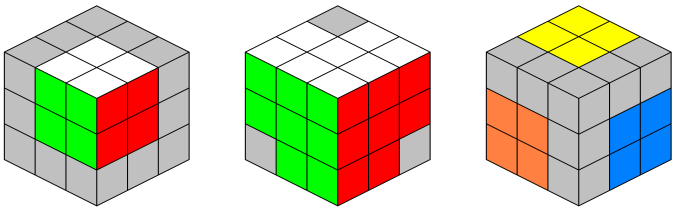


Figure 4.6: Tripod block-style

Because it is comparable to CrossF2L and BlockF2L and in order to demonstrate the technique of combining statistics, I let Hume solve 300 cubes with

this method as well. Table 4.5 shows the results to the level of average step lengths. The left column is the actual outcome produced by Hume. Because of Hume's unconditional method specifications, I had to choose a fixed initial subcube. After that, I can tell Hume to solve the three extensions in any order.

Tripod fixed start		Tripod free start		BlockF2L	
Initial Subcube	5.97	Initial Subcube	**5.27**	Block 1	**5.27**
Extension 1	5.22	Extension 1	5.22	Block 2	5.49
Extension 2	6.21	Extension 2	6.21	Block 3	6.20
Extension 3	7.86	Extension 3	7.86	Block 4	6.79
Total	25.26	Total	**24.56**	Total	23.74

Table 4.5: Tripod with fixed/free start, BlockF2L

But I wanted to allow the initial subcube to be chosen just as freely as for the BlockF2L method. For this, I simply "plugged in" the statistic from the first block of BlockF2L and adjusted the "total" value, shown as the middle column in the table. As the first step of both methods is intended to be the same, this is correct for the first step of the intended Tripod method. After that initial subcube has been solved, the second step is to "extend an already solved subcube" and at that point it doesn't matter anymore whether that subcube was fixed or freely chosen, so the results for the remaining steps are correct as well. This technique thus allows emulating freedom.

Have a quick look again at the table and compare the right two columns. The first extension towards the tripod takes fewer turns than the first extension towards full two layers. That's because three adjacent extension points are better than two plus one diagonal (as mentioned earlier, adjacent extension is easier than diagonal). Next, both methods have already built a 2x2x3 block and offer two possible extensions. For BlockF2L the extensions overlap slightly and for Tripod they don't, but this is insignificant, hence the almost identical average length of the third step (and the difference can also be partly attributed to the random nature of the experiment). The fourth step of the Tripod method takes longer, obviously because solving three pieces is harder than solving just two.

Chapter 5

Implementation of Hume

This chapter describes the inner workings of Hume, which can be roughly divided into three parts:

1. Parse the input.
2. Prepare auxiliary lookup tables.
3. Carry out the requested tasks, i.e., repeatedly scramble and solve the puzzle.

Parsing the input and scrambling the puzzle are rather trivial and obvious, but preparing the lookup tables and solving a scrambled puzzle are described next.

5.1 Preparation

Hume's search doesn't work with individual pieces but at the subgoal level. Remember a subgoal is a set of pieces to solve. A "subgoal state" describes the current whereabouts of those pieces, and these subgoal states are what Hume works with. It needs to know how far a subgoal state is from being solved and needs to apply turns to subgoal states. It therefore prepares these two lookup tables for each subgoal:

Distance(SubgoalState) The minimum number of turns required to solve the subgoal from the given state.

Successor(SubgoalState, Turn) The successor state of applying the turn to the state.

These are simply computed by a breadth-first search started at the "solved" subgoal state. Note that for building the tables Hume does operate on individual pieces/stickers using the **puzzle** part of the input, but during the process it encodes a whole subgoal state as one integer. The later search then operates on these numbers, and the distance and successor tables are simple numerically indexed arrays.

As this can take some time, the data is stored in the `subgoals` directory so that subsequent runs of Hume can reuse it rather than recompute it, which is much faster.

5.2 The solving algorithm

After scrambling, Hume solves the puzzle in a two-layer search described next. The outer layer works at the method level, arranging which subgoals are to be solved next, but not actually computing solutions for them. That is the job of the inner layer, which is called repeatedly from the outer layer.

5.2.1 The outer layer search

The outer layer arranges which subgoals need to be solved next, and calls the inner layer to receive actual solutions for them. The outer layer thus only needs to think in terms of method steps and their subgoals. It solves the puzzle according to the entered solving method, solving more and more subgoals. Besides remembering at which method step it is, it holds the current situation of the search as:

1. The current state of the puzzle.
2. What has been solved already (a set of subgoals).
3. What shall be solved next (a set of subgoals).

```
SolvedSubgoals = empty set
CurrentSubgoals = empty set
for each method step
    CurrentSubgoals = the subgoals of the method step
    while CurrentSubgoals not empty
        solve at least one of CurrentSubgoals
        move solved CurrentSubgoals to SolvedSubgoals
```

Above is pseudo code for the outer layer. Consider solving Rubik's cube with the CFOP method. For the first step, the set of already subgoals is empty and the set of subgoals to be solved next contains just one subgoal, namely to solve the white cross. After the cross is solved, the set of already solved subgoals contains just the "cross" subgoal. The set of subgoals to be solved next contains the four corner-edge pair subgoals. Hume then solves one of these four subgoals (in addition to keeping the cross subgoal solved). After one of those four subgoals is solved, it is moved from the "to be solved" set to the "solved already" set. This process is then repeated until the "to be solved" set is empty and all its subgoals have moved to the "solved already" set. Then the next method step is approached.

Besides the HTML output containing overall statistics and the found solutions, Hume prints log messages to the standard error stream to tell the user what the program is currently doing. An example log output for a CrossF2L solve is shown below, exemplifying how the outer layer search introduces and handles the method steps and their subgoals.

```
Solve 3:
-------------------------------------------
We're about to...
  keep these subgoals solved:
  solve at least one of these subgoals:
    WO WB WR WG  (=187186)
-------------------------------------------
Found solution: [D1 F1 R3 B3 D1 F2 R1]
Visited 714 states

-------------------------------------------
We're about to...
  keep these subgoals solved:
    WO WB WR WG
  solve at least one of these subgoals:
    WOB OB WO WB  (=35810)
    WGO GO WG WO  (=47757)
    WRG RG WR WG  (=35507)
    WBR BR WB WR  (=179536)
-------------------------------------------
Found solution: [U1 F1 L2 F1 L2 F3]
Visited 2580 states
```

```
-----------------------------------------
We're about to...
  keep these subgoals solved:
    WO WB WR WG
    WRG RG WR WG
  solve at least one of these subgoals:
    WOB OB WO WB  (=20305)
    WGO GO WG WO  (=49919)
    WBR BR WB WR  (=132017)
-----------------------------------------
Found solution: [U1 F1 L1 F3 L3 F3]
Visited 2076 states

-----------------------------------------
We're about to...
  keep these subgoals solved:
    WO WB WR WG
    WRG RG WR WG
    WGO GO WG WO
  solve at least one of these subgoals:
    WOB OB WO WB  (=168412)
    WBR BR WB WR  (=179960)
-----------------------------------------
Found solution: [R1 U1 F1 R1 F3 R3]
Visited 270 states

-----------------------------------------
We're about to...
  keep these subgoals solved:
    WO WB WR WG
    WRG RG WR WG
    WGO GO WG WO
    WOB OB WO WB
  solve at least one of these subgoals:
    WBR BR WB WR  (=198708)
-----------------------------------------
Found solution: [R2 U2 R1 B2 L3 B3 L1 B3 R1]
Visited 281193 states

The whole solution (34 moves):
D1 F1 R3 B3 D1 F2 R1
U1 F1 L2 F1 L2 F3
U1 F1 L1 F3 L3 F3
R1 U1 F1 R1 F3 R3
R2 U2 R1 B2 L3 B3 L1 B3 R1
```

5.2.2 The inner layer search

The inner layer finds an algorithm that solves at least one of the subgoals to be solved next, while keeping the previously solved subgoals solved. Of course the latter don't need to be kept solved during every turn, but must be solved at the end of the found solution algorithm. The inner layer gets exactly what the outer layer manages:

1. The current state of the puzzle.
2. What has been solved already (a set of subgoals).
3. What shall be solved next (a set of subgoals).

It receives the state of the whole puzzle as a starting point for the search, but it doesn't actually work with it. It only uses the given whole puzzle to initialize the search, by extracting the subgoal states for all given subgoals (both solved and current). From then on, the search only works with those subgoal states rather than a whole puzzle state, and their combination makes up a search state or "node" in terms of a graph search.

To apply a puzzle turn to such a node, the turn is applied to each of the subgoal states by using their respective **Successor** tables. To determine how far away a node is from "solved", the values from the respective **Distance** tables are combined in a way shown soon.

Note that while the inner layer receives several subgoals that shall be solved, it doesn't need to solve all of them but only one of them. As discussed earlier, it shall be the one that requires the fewest turns to be solved. But how do we find out which one that is?

The straight-forward approach simply tries them serially, one after the other. For example, if there are already solved subgoals S_1, S_2, S_3 and current subgoals C_1, C_2, C_3, we could try solving $S_1+S_2+S_3+C_{\mathbf{1}}$, then $S_1+S_2+S_3+C_{\mathbf{2}}$, then $S_1 + S_2 + S_3 + C_{\mathbf{3}}$. We would get one solving algorithm for each case and return the shortest.

Parallel search

However, this is rather slow because we'd do three complete searches. A faster approach is to try all three searches in parallel and stop as soon as one succeeds. If let's say C_1 needs only three turns to be solved and the

other two need nine turns each, then we'll only do three parallel searches to depth three rather than one to depth three and two to depth nine.

Hume employs the A* graph search algorithm and does search in parallel. Although rather than several independent parallel searches it does one combined parallel search. In the example, that's one combined search for all of $S_1, S_2, S_3, C_1, C_2, C_3$ and a graph node holds the combination of all six subgoal states. The heuristic that A* needs as lower bound for how far such a node is from "solved" is determined from the **Distance** values as follows:

$$\text{maxDist}(\ S_1, S_2, S_3, \text{minDist}(\ C_1, C_2, C_3\ ))$$

For solving at least one of C_1, C_2, C_3, we certainly need at least as many turns as for the easiest of them, which explains the minDist part. For solving all of S_1, S_2, S_3 and the easiest C, we certainly need at least as many turns as for the hardest of them, which explains the maxDist part.

To test the efficacy of the combined parallel search compared to the serial search, I tried both of them with the CrossF2L and the BlockF2L method on 1000 solves each. Table 5.1 shows the average number of nodes visited during the search as well as the average time. Examine the CrossF2L results first. The cross visits about the same number of nodes both in the serial and the parallel search, which is expected as there is no parallelism. Pairs 1 to 3 benefit a lot from the parallel search, but the last pair again has no parallelism and thus can't benefit. Unfortunately that last pair is by far the main contributor and dominates the other steps, far reducing the overall advantage of parallel over serial search. The BlockF2L method showed the same behavior. The conclusion is that parallel search does help, but sadly much less than I had hoped.

CrossF2L		
	serial	parallel
Cross	314	338
Pair 1	8127	1155
Pair 2	8550	1698
Pair 3	12677	3504
Pair 4	27559	28562
Total	57227	35256
Seconds	0.83	0.62

BlockF2L		
	serial	parallel
Block 1	1160	327
Block 2	9388	1000
Block 3	18511	4427
Block 4	24687	24974
Total	53746	30727
Seconds	0.66	0.46

Table 5.1: Visited nodes in serial/parallel search

A possible bug

The combined parallel search works well, but while writing chapter 4 which analyzes the results, I noticed a deviation which I believe hints at a subtle but noticeable bug. The fourth corner-edge pair in the CFOP method and the fourth block in the Block method are equivalent, i.e., that fourth block *is* a corner-edge pair. They thus ought to require the same number of turns on average, but didn't.

In ten thousand solves, the fourth block took 6.76 on average while the fourth pair took 6.84 turns, despite being equivalent. Of course the results are of experimental nature and thus some difference is to be expected, but I feel this is too large and I had noticed consistent and even slightly larger differences in smaller sample sets earlier. I then reran the experiment with the slower serial search described above and found much closer averages of 6.79 and 6.81 turns (as shown earlier in table 4.2).

My educated guess is that the use of A* with the combined lower bound heuristic causes an undesired synergy effect between the not yet solved subgoals. The A* algorithm prioritizes nodes that are close to a goal, expanding them first. This way, a subgoal close to solved can somewhat "pull" the search in its direction. Now if optimal solutions for two subgoals happen to begin the same way, they might combine their pulling strengths to make A* search in their common direction. And while probably only one of them ends up solved, the other is left "almost solved" and gets solved in few turns in the next search.

If this theory is correct, it can account for the fourth block needing fewer turns than expected. Why the fourth pair needs more than expected is unclear, I can only presume an adverse effect and theorize that two neighboring blocks may profit from having a shared edge while two pairs don't profit because they don't share common pieces.

The discrepancy is rather small, but it is still undesired. The conjectured synergy doesn't reflect how human speedcubers actually work and thus betrays Hume's purpose of simulating them. I will therefore change the implementation to not use one combined parallel search but several independent parallel searches. This should provide the more accurate results of independent serial searches while still being faster.

Unfortunately I realized this problem late in the process and I'm not able to correct it before my thesis deadline. The results shown in chapter 4 are taken from the more accurate serial searches, and future versions of Hume will be available on the accompanying website mentioned in the foreword.

On the other hand, if the objective is not to reflect human solving but to find short solutions, then exploiting and possibly even amplifying the effect would be welcome. Also, in the fewest moves competitions we have enough time to think and search for solutions, so there this might turn out to be a viable technique. Finally, even in speedcubing with the CFOP method there's an approach called "multi-slotting" which attempts to solve one corner-edge pair in a way to make the next pair easier. But it looks at the next pair only close to the end of solving the current pair, not already at the start like I conjecture Hume does. My conclusion for now is that I want to study both ways more, the careless individual way as well as the considerate synergetic way.

Chapter 6

Future

This chapter describes possible future activities for improving and using Hume, many of which I intend to undertake. On the improvement side, more functionality could be introduced and existing functionality could be made more efficient. The usage side refers to applying Hume to more puzzles and more methods.

6.1 Analyzing more puzzles and methods

Chapter 1 already showed a small collection of puzzles. There are dozens more, if not hundreds. Many are theoretically equivalent and are just shaped differently on the outside, but there are also many fundamentally different puzzles. As an example of a rather strange puzzle, figure 6.1 revisits the Skewb puzzle. Its shape is a cube, but it has four (!) twisting planes, shown by the thick black lines. Each twisting plane diagonally cuts the puzzle in half.

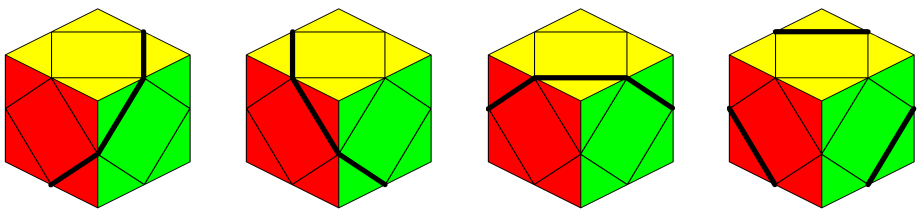


Figure 6.1: The four ways to turn the Skewb

I want to analyze more puzzles with Hume than just the 3x3x3, at least the most popular including my personal favorite, the Megaminx. Also, there

are several more methods for solving the 3x3x3 than analyzed in chapter 4 that I'd like to study.

6.2 Orient first, grouping

Many methods at some point orient first or permute first. That is, a subgoal doesn't necessarily solve pieces completely. It can also just solve them partially and defer the other part to a later subgoal. Permutation-first is already possible in Hume with a little trick, by extending the puzzle description in the input. Let's say after solving two layers, the next subgoal shall be to permute the last layer edges without caring about their orientation. This can easily be encoded by adding twelve "imaginary" orientation-less edges to the puzzle definition in the input, which get moved simultaneously with the real edges. Then the subgoal, instead of requesting the real edges to be permuted, requests their imaginary counterparts to be solved.

So permutation-first is possible already, albeit in a rather artificial way. Orientation-first however isn't possible yet at all. This is because Hume right now works on permutations of unique elements. A solution would be to get rid of this uniqueness restriction and allow a subgoal definition like "Place the four yellow corner stickers on these four places, but I don't care which goes where". This would also allow natural handling of for example the 4x4x4 cube, which does have real pieces that are indistinguishable (each side has four identical center pieces, see figure 6.2).

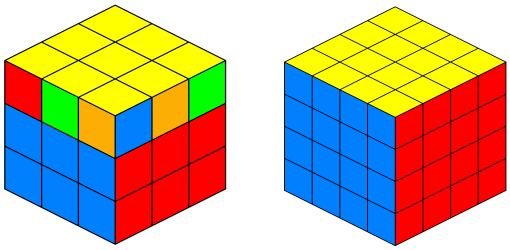


Figure 6.2: Orienting the last layer, 4x4x4 cube

I expect this to be rather easy to implement, lexicographical sorting of the sticker locations of a subgoal should suffice, as that would ignore the order and offer a good representative for each set of locations.

6.3 Movement restrictions

There are puzzles for which not all turns are possible at all times. The ”bandaged cube” shown in figure 6.3 comes to mind, although it is a bit artificial and while it’s an intriguing puzzle, it’s sadly not particularly popular. However, the Square-1 puzzle also has a serious movement restriction due to its different piece shapes and is popular enough to be among the few puzzles that appear in the speedcubing competitions. Hume so far offers no way to describe such restrictions, i.e., to specify when a turn is possible and when it’s not.

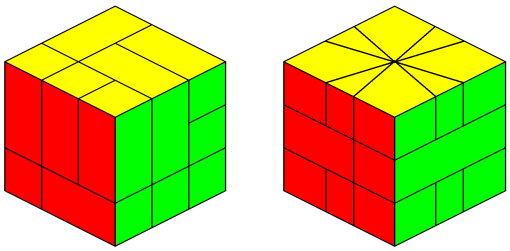


Figure 6.3: Bandaged cube and the Square-1

6.4 More accurate measurements

For regular speedcubing, in the end all that matters is time. Fewer turns or fewer method steps usually lead to faster times, but this is not the whole story. In reality, some turns like B2 are rather awkward to execute while others are easy. The same is true not just for single turns but for algorithms as well. For example, URU’ is considerably easier to do fast than URU, even though they differ only in the direction of the last turn. When doing UR’U, one would pull U with the right index finger, then turn R by rotating the right hand. At that point, the fingers are in an awkward position for the second U, no finger can turn it immediately. In contrast, the final U’ in URU’ can simply be pulled with the index finger of the left hand. Another example is that while a single B turn is awkward to do alone, it’s quite alright inside the algorithm RBR’. The first R brings the right hand in a position to simply pull B with the right index finger.

In order for an algorithm to be fast, it should thus be ”finger-friendly”. Ultimately, this will require the program to somehow represent where the fingers are and how fast certain turns and their combinations can really be executed. I don’t know any programs that do this, but it would be very helpful and I expect this to happen in the near future.

Notice however that execution speed is still not the whole story. That's because along the solve, one has to recognize the situation and decide how to go on. This causes breaks, moments when one doesn't execute any turns. Easier recognition is thus beneficial for faster overall solve times. It is probably very hard to estimate in an automated way, though, as it depends on the specific techniques of how cubers recognize the cases.

6.5 Memory usage

Currently, Hume uses a lot of memory which in many cases can be reduced considerably with algorithmic improvements and by using less wasteful data structures. There are four data structures which make up most of the memory usage.

6.5.1 The Successor tables

Different subgoals often have the same number and types of pieces. For example, the four corner-edge pairs of the CFOP method all contain one corner and one edge. With the optimization of adding the two adjacent cross edges, they all contain one corner and three edges.

While the subgoals of different corner-edge pairs do talk about different pieces, their successor tables are actually isomorphic. Remember that this table tells how the turns move pieces around. But the turns don't care what pieces they move. If a turn takes some pieces from A to B, then other pieces at A would also be brought to B by the same turn.

So if there are several subgoals with the same number and types of pieces, all could use the same successor table rather than each having its own. This would already help for the 3x3x3 cube, but a lot more for the Megaminx puzzle shown in figure 6.4, which I would like to tackle as well. It's structurally very similar to the 3x3x3 cube in that it has centers, edges and corners. It just has more of everything. I would like to analyze it with Hume and would have a lot of block-style subgoals which could share one successor table. Compared to the 3x3x3 cube, there are many more subgoals, each subgoal has many more possible states, and there are twice as many turns. Without sharing a successor table, the total memory requirement would likely make the intended Hume analysis of the Megaminx infeasible even on modern PCs.

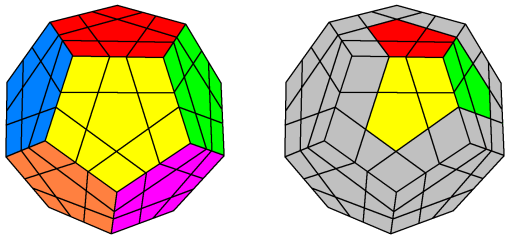


Figure 6.4: Megaminx and a block-style subgoal

6.5.2 The Distance tables

Like the successor tables, the distance tables could be shared among several subgoals rather than each subgoal having its own separate table. However, this is possible in fewer cases, harder to do, and wouldn't save nearly as much as sharing successor tables because they're much smaller anyway.

Successor tables can be shared among any subgoals with the same number and types of pieces, e.g., subgoals dealing with one corner and three edges. This is not the case for the distance tables. Have a look at the two subgoals in figure 6.5, both covering one corner and three edges.

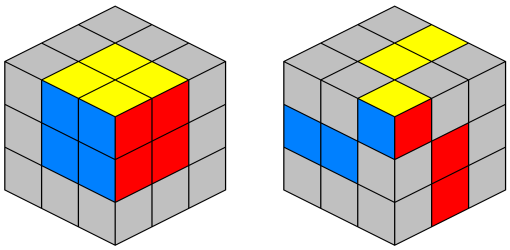


Figure 6.5: Subgoals with incompatible distance tables

For both subgoals, there is one state with distance zero. The left goal has nine states at distance one, reachable with any U, R or F turn. The right subgoal however has 18 states at distance one, reachable with any turn of any side. This already shows that the two distance tables are fundamentally different and can't be shared.

6.5.3 A* queue and visited states

The successor and distance tables are fully set up during the startup of Hume, before the actual search begins. During the search, the memory consumption rises considerably higher. This is due to the A* queue of states

to visit, as well as the set keeping track of the already visited states. Both can be reduced by a probably significant constant factor by using more economical data types, preferably eliminating objects since Java seems to have a serious overhead for them. An alternative would be to port the program to a different language. A good option might be C++, which also has typedefs so that variables of a primitive type like ”int” could still be declared as something like ”SubgoalState” without the performance drawbacks of objects.

6.6 Small subgoals

The end of chapter 3 introduced the idea of overlapping subgoals, one benefit being the search getting much faster thanks to the much better heuristics of large subgoals compared to small ones. Of course this can be a rather unnatural and undesired way to specify the method. For CrossF2L I'd really prefer to just specify the cross and the four corner-edge pairs, rather than artificially extending the pairs. The user of the program ought not have to know and think about such pitfalls of the implementation. A possible solution would be to automatically combine small subgoals once they're solved.

Chapter 7

Conclusion

I already consider Hume a success, since it has made the analysis carried out in chapter 4 quite easy and has generated useful statistics and solutions. As intended, specifying the methods was fairly straightforward and the high level of abstraction required little work. The statistical results as well as studying individual solves were very interesting for me and I hope others will find them valuable as well. I believe by spending more time with it, my knowledge and expertise concerning solving the cube and other puzzles will grow.

As discussed in the previous chapter, improving the existing implementation and adding more functionality should make Hume even much more helpful. I see a bright future for it and possibly other new programs working on the method level. Maybe one day we will take this another step further, where no method is entered by a human for the computer to evaluate, but where the computer autonomously ”invents” and evaluates methods, so that many more methods can be tested than just those we humans come up with.

On the other hand, while the computer scientist in me is very excited about this future, the cuber and human in me is somewhat terrified, because the creative activity of inventing new methods and discussing them with others is one of the most fascinating aspects of cubing. I do not want to be degraded to following the instructions of machines. I'd rather like this to stay the other way around.

Bibliography

[1] J. Fridrich. My speed cubing page, 1997. URL http://www.ws.binghamton.edu/fridrich/cube.html

[2] Bernard Helmstetter. Rubik's cube : Algorithms for the last layer, 2001. URL http://www.ai.univ-paris8.fr/~bh/cube/

[3] Dan Hoey. Re: lower bounds, 1981. URL http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Dan_Hoey__Re__lower_bounds.html

[4] Stan Isaacs. lower bounds, 1981. URL http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Stan_Isaacs__lower_bounds.html

[5] Josef Jelinek. Rubik's cube solution methods. URL http://software.rubikscube.info/

[6] Herbert Kociemba. Cube explorer, . URL http://kociemba.org/cube.htm

[7] Herbert Kociemba. Two-phase-algorithm and god's algorithm, . URL http://kociemba.org/performance.htm

[8] Herbert Kociemba. Some thoughts about a proof, that 24 moves suffice, 2008. URL http://cubezzz.homelinux.org/drupal/?q=node/view/115

[9] R. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the Workshop on Computer Games (W31) at IJCAI-97*, pages 21–26, Nagoya, Japan, 1997. URL citeseer.ist.psu.edu/korf97finding.html

[10] Daniel Kunkle and Gene Cooperman. Twenty-six moves suffice for rubik's cube. In *ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 235–242, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-743-8. doi: http://doi.acm.org/10.1145/1277548.1277581

[11] Bruce Norskog. The 4x4x4 can be solved in 79 moves (stm), 2006. URL http://cubezzz.homelinux.org/drupal/?q=node/view/62

[12] Bruce Norskog. The 4x4x4 can be solved in 77 single-slice turns, 2007. URL http://cubezzz.homelinux.org/drupal/?q=node/view/93

[13] Silviu Radu. How to compute optimal solutions for all 164,604,041,664 symmetric positions of rubik's cube, 2006. URL http://cubezzz.homelinux.org/drupal/?q=node/view/63

[14] Silviu Radu. Rubik can be solved in 27f, 2006. URL http://cubezzz.homelinux.org/drupal/?q=node/view/53

[15] Silviu Radu. Solving rubik's cube in 28 face turns, 2005. URL http://cubezzz.homelinux.org/drupal/?q=node/view/37

[16] Michael Reid. two stage filtration, 1995. URL http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__two_stage_filtration_(2).html

[17] Michael Reid. superflip requires 20 face turns, 1995. URL http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__superflip_requires_20_face_turns.html

[18] Michael Reid. new upper bounds, 1995. URL http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__new_upper_bounds.html

[19] Tomas Rokicki. Contest to find a short program to solve the cube. URL http://tomas.rokicki.com/cubecontest/

[20] Tomas Rokicki. In search of: 21f*s and 20f*s; a four month odyssey., 2006. URL http://cubezzz.homelinux.org/drupal/?q=node/view/56

[21] Tomas Rokicki. Twenty-five moves suffice, 2008. URL http://cubezzz.homelinux.org/drupal/?q=node/view/112

[22] Jaap Scherphuis. Computer puzzling, . URL http://www.geocities.com/jaapsch/puzzles/compcube.htm#thisal

[23] Jaap Scherphuis. Thistlethwaite's 52-move algorithm, . URL http://www.geocities.com/jaapsch/puzzles/thistle.htm

[24] Lars Vandenbergh. Imagecube, 2004. URL http://www.cubezone.be/imagecube.html

[25] Dik Winter. Corrected calculations are now done., 1992. URL http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Dik_T._Winter__Corrected_calculations_are_now_done..html

[26] Dik Winter. Kociemba's algorithm, 1992. URL http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Dik_T._Winter__Kociemba's_algorithm_(3).html

Appendix A

Lower bounds from counting arguments

A simple counting argument[4] already shows that there are cube states that require at least 17 turns to solve. It works the opposite direction, though, showing that there are cube states that require at least 17 turns to be *reached* from the solved state. So we start at the solved cube and scramble it, adding one turn at a time. The first turn can produce exactly 18 new states, one for each possible turn (six sides times three angles). Subsequent turns can produce at most 15 times as many new states as the previous turn, because turning the same side twice in a row doesn't yield new states. The *adding single turns* part of table A.1 displays the numbers, showing that 16 turns are not enough to produce all $4.3 \cdot 10^{19}$ states, but 17 turns are.

A more elaborate counting argument[3] is able to prove that 18 turns are needed, by avoiding trivial wasteful sequences of turns of opposite sides. For example, LRL'R' has no effect at all, and FB and BF have the same effect so only one should be counted. Thus again starting with the solved cube, we don't add single turns but instead segments of combined opposite layer turns. For example:

(U) (R' L2) (D') (F) (D2 U') (R) (F' B) (D)

For each pair of opposite layers, there are six ways to make *one* turn (choice of two layers times choice of three angles) and nine ways to make *two* turns (three angles for one side times three angles for the other side). Now let S_N be the number of scrambles with N turns. The first segment we apply to the solved cube can choose between three pairs of opposite sides (UD, LR, FB). Subsequent segments only have a choice between the two pairs not being the previous segment, as using the same pair twice in a row would again be

	Adding single turns		Adding turn segments	
Turns	New	Total	New	Total
0	1	1	1	1
1	18	19	18	19
2	270	289	243	262
3	4,050	4,339	3,240	3,502
4	60,750	65,089	43,254	46,756
5	911,250	976,339	577,368	624,124
6	13,668,750	14,645,089	7,706,988	8,331,112
7	$2.0503 \cdot 10^{08}$	$2.1968 \cdot 10^{08}$	$1.0288 \cdot 10^{08}$	$1.1121 \cdot 10^{08}$
8	$3.0755 \cdot 10^{09}$	$3.2951 \cdot 10^{09}$	$1.3732 \cdot 10^{09}$	$1.4845 \cdot 10^{09}$
9	$4.6132 \cdot 10^{10}$	$4.9427 \cdot 10^{10}$	$1.8331 \cdot 10^{10}$	$1.9815 \cdot 10^{10}$
10	$6.9198 \cdot 10^{11}$	$7.4141 \cdot 10^{11}$	$2.4469 \cdot 10^{11}$	$2.6450 \cdot 10^{11}$
11	$1.0380 \cdot 10^{13}$	$1.1121 \cdot 10^{13}$	$3.2662 \cdot 10^{12}$	$3.5307 \cdot 10^{12}$
12	$1.5570 \cdot 10^{14}$	$1.6682 \cdot 10^{14}$	$4.3599 \cdot 10^{13}$	$4.7129 \cdot 10^{13}$
13	$2.3354 \cdot 10^{15}$	$2.5023 \cdot 10^{15}$	$5.8198 \cdot 10^{14}$	$6.2911 \cdot 10^{14}$
14	$3.5032 \cdot 10^{16}$	$3.7534 \cdot 10^{16}$	$7.7685 \cdot 10^{15}$	$8.3976 \cdot 10^{15}$
15	$5.2547 \cdot 10^{17}$	$5.6301 \cdot 10^{17}$	$1.0370 \cdot 10^{17}$	$1.1209 \cdot 10^{17}$
16	$7.8821 \cdot 10^{18}$	$\mathbf{8.445 \cdot 10^{18}}$	$1.3842 \cdot 10^{18}$	$1.4963 \cdot 10^{18}$
17	$1.1823 \cdot 10^{20}$	$\mathbf{1.267 \cdot 10^{20}}$	$1.8477 \cdot 10^{19}$	$\mathbf{1.997 \cdot 10^{19}}$
18	$1.7735 \cdot 10^{21}$	$1.9001 \cdot 10^{21}$	$2.4664 \cdot 10^{20}$	$\mathbf{2.666 \cdot 10^{20}}$

Table A.1: Overestimated numbers of states reachable

useless. These thoughts lead to the recurrence relation shown below. The *adding turn segments* part of table A.1 displays the numbers, showing that 17 turns are not enough to produce all $4.3 \cdot 10^{19}$ states, but 18 turns are.

$$\begin{aligned} S_0 &= 1 \\ S_1 &= 3 \cdot 6 \cdot S_0 \\ S_2 &= 2 \cdot 6 \cdot S_1 + 3 \cdot 9 \cdot S_0 \\ S_{N \geq 3} &= 2 \cdot 6 \cdot S_{N-1} + 2 \cdot 9 \cdot S_{N-2} \end{aligned}$$

Note that besides trivial wasteful things like turning the same side twice in a row, or LR and RL having the same effect, there are also longer, nontrivial sequences having equal effects. For example, R2 B' R' B U and U F R' F' R2 do exactly the same, but the above counting arguments count them separately. The numbers in table A.1 are therefore still an overestimation of the cube states reachable with a certain number of turns. One can not conclude that all cubes can be solved in 18 turns just because there are more algorithms of this length than there are cube states, because many duplicates are produced. These counting arguments thus only provide lower bounds (for the number of turns needed to reach/solve all states).