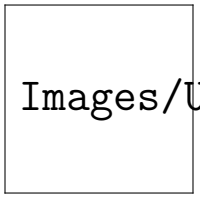


University of Reading

Department of Computer Science

Computer Science Undergraduate Report - Rubik's Cube Solver

Callum Claton David McLennan

Supervisor: Martin Lester

A report submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Bachelor of Science in *Computer Science*

February 23, 2021

Declaration

I, Callum, Claton, David, McLennan, of the Department of Computer Science, University of Reading, confirm that all the sentences, figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

Callum David Claton McLennan
February 23, 2021

Abstract

..

Keywords: Rubik's Cube,

Report's total word count:

Glossary

This is standard Rubik's Cube notation[**notation**] that only applies to 2x2x2 and 3x3x3 cubes.

Terminology

Cubie	One of many smaller cubes that make up the Rubik's cube.
Center	A cubie with one colour on the face in the center of the cube.
Edge	An edge cubie has two colours as they're on the edge of the cube.
Corner	A corner cubie has 3 colours and there are always 8, regardless of cube size.
Face	A face is a side of a Rubik's Cube. There are 6 faces regardless of size.
A letter by itself refers to a clockwise rotation of a single face by 90.	
A letter with a ' ' ' is a 'prime move' which means the face rotates counter-clockwise 90.	
A letter with the number 2 after it marks a double turn 180.	
X, Y, Z rotations aren't normally required to solve a cube. These are whole cube rotations.	

Moves

	Front	Right	Up	Left	Back	Down	Cube Rotation		
Clockwise Moves	F	R	U	L	B	D	X	Y	Z
Counter-Clockwise Moves	F'	R'	U'	L'	B'	D'	X'	Y'	Z'
Double Moves	F2	R2	U2	L2	B2	D2	N/A		

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Rubik's Cube[**optimalalgos**] is stereo-typically a really difficult puzzle to solve. There are various methods in solving it but for any average person it takes countless hours of learning to be able to solve it just once. That's where a Rubik's Cube Solver comes in; solving the cube using various human and computing methods. This paper will represent the various methods I tried, what was more efficient/less efficient and the issues I encounter throughout the creation process. I will be adding what motivated my decisions regarding the features and methods I include.

1.1 Background

The Rubik's Cube is a well known puzzle all around the world and it's considered extremely difficult to solve. The standard sized 3x3x3 cube has a whopping 4.3 quintillion different permutations, a 4x4x4 has 7.4 quattuordecillion (46 digit number) permutations... You can appreciate the reason for the development and use of algorithms/methods of solving cubes of these sizes and larger.

1.2 Aims and Objectives

Albeit an ambitious goal, I hope to create a solver using a heuristic based search algorithm or constraint solver. Depending on how well I make progress, I would also hope to solve the 3x3x3 cube and bigger cubes as the differences are the centers that need to be solved and the increase in moves that are needed to solve it. It's very likely I could stray from what I've outlined here and use a different, more efficient and well suited algorithm to solve the cube(s).

A cool feature I'm eager to implement is a user specified custom scramble (Hopefully correlating with a real physical cube) and get my program to solve it, then provide the algorithm (steps) to solving that particular scramble.

1.3 Research Hypothesis

Although this project is ambitious, I'm hoping that with my current experience solving Rubik's Cubes, this project will progress more fluently than prior projects I've pursued and hopefully I'm able to better diagnose, understand and solve issues during the project creation.

I'm excited to discover if I can create or piece together an algorithm that is versatile enough to solve Rubik's Cubes of various sizes. Considering that I feel this project is a big challenge just solving the 3x3x3 sized cube, I wouldn't be too disappointed if I couldn't manage to create an algorithm that can solve all sized cubes.

Chapter 2

Literature Review

2.1 Rubik's Cube Emulator

I discovered a Rubik's Cube Emulator tutorial online, fortunately in my chosen language. There are three parts to the series, the first[**em1**] covering the visual representation of the cube made Cubie objects. The second part[**em2**] guided me through creating the rotations and adding a "Face" class representing each Cubie sticker. The third part[**em3**] covered the animation for each move applied to the Rubik's Cube. This code structure should be relatively easy to follow and interpret; a big reason I chose the processing-java language.

Since following those tutorials, I've catered the emulator to produce smaller and bigger cube sizes, as well as create every possible move for each cube (Although they're not all accessible through the keyboard as you get bigger than a 3x3x3 since there's limited notation). I've also added a 2D representation of the cube within the GUI and hope to make the cube state modifiable by the user. Furthermore, I've refined the program to only display/store the Cubie objects that are visible to the user - it does not store the Cubies that would otherwise be within the cube. This saves the CPU a lot of stress when generating the larger cube sizes.

2.2 Review of Current Best Methodologies

After some research, I've discovered that there are various preexisting algorithms that efficiently solve a 3x3x3 Rubik's Cube in 20 moves or less, often regarded to as "God's Number"[**godsnumber**].

2.2.1 Multi-Phase Algorithms

Thistlethwaite's Algorithm

This method was one of the first multi-phase computing solving algorithms for solving the Rubik's Cube. This algorithm reduces the scrambled cube into sub-problems/groups which acts as a heuristic for the Iterative Deepening Depth First Search algorithm used to search for a solution for each corresponding stage.
The groups are typically identified as **G0, G1, G2, G3 and G4**.

Figure 2.1: G0 is any scrambled cube.

Figure 2.2: G1 is when all the edges are oriented according to set rules [**erules**]

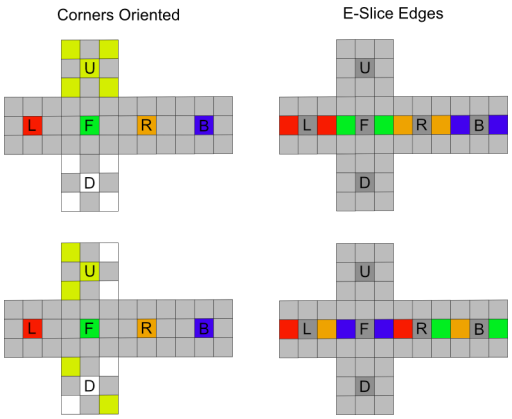


Figure 2.3: G2 is the orientation of all corners and the edges that belong in the E slice are within the e-slice. Permutation/Orientation within the slice doesn't matter.

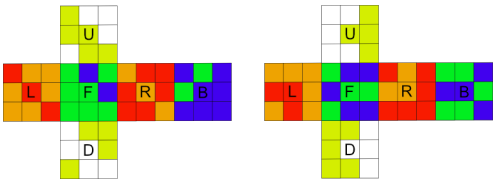
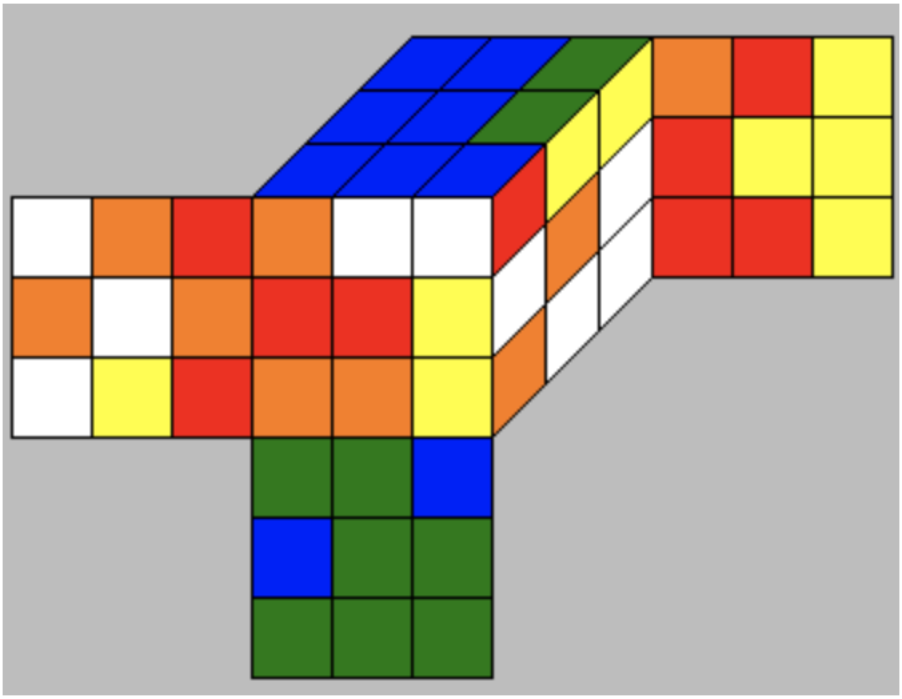


Figure 2.4: G3 is when permutation of corners/edges such that colours of these cubies are on the correct faces/opposing face to their correct faces.

Kociemba's Algorithm

The cube needs to be put into a "G1 State" as in Figure 2.1.

Figure 2.5: Kociemba's Algorithm - G1 State



The G1 state adheres to the following three rules

1. Depending on what faces are regarded as the U and D faces, the corners are oriented so their stickers correlate with either the U and D faces on the U and D faces. (Even if they're in the wrong location)
2. All the edges are oriented (U/D faces will have the corresponding stickers).
3. All the middle layer edges oriented in a similar fashion to the prior edges but the stickers are on the face they belong on the opposing face, else they're oriented incorrectly. (Ready for swapping/re-orienting)

Korf's Algorithm

The most optimal solving algorithm as it finds the solution in 20 moves or less. This is a really efficient algorithm in regards to moves but not time.

This method uses IDA* search and doesn't rely on breaking down the cube into sub-problems; it relies heavily on pruning tables. Pruning tables contain information that the program uses to determine whether a branch in the search tree is worth looking at. If not, it skips it - saving considerable amounts of time.

Pruning tables generated through a breadth first search that performs combinations of moves on a cube and records the number of moves it required to get to that state. This means, when the algorithm is running, it sends the cube's state after testing some moves to a decoding function which returns an index representing which line in the byte file containing the value representing the number of moves to get to a solved state. If the number is more than the depth we're looking it, we skip searching down that branch.

2.3 Reduction Method

This method is primarily for cubes larger than 3x3x3. It involves reducing the scrambled cube down to three primary problems. Solving the 'center' cubies, solving the 'edge' cubies and finally solving the entire cube like a normal 3x3x3 cube.

2.4 Critique of the Review

2.5 Description of Project in Context of Existing Literature

2.6 Metrics

2.7 Summary

From what I've researched and experimented with myself, I think Kociemba's algorithm is the most reasonable and practicable algorithm to use as it solves cubes within a reasonable amount of moves without taking too long to solve.

Chapter 3

Methodology

3.1 Programming Language

The programming language of choice is Processing-Java[REFERENCE] as the language for this project as I already have some experience using this language. It's quite intuitive to use, has tutorials regarding Rubik's Cube emulators and automatic memory management.

I started off using the Processing IDE[REFERENCE] but discovered random bugs/issues when trying to debug/run my program which greatly lagged my computer so I switched over to using VS-Code (a code editor) and imported the java-processing libraries in order for me to improve performance and run my program.

This isn't the fastest programming language to use when compared to C/C++ but I don't think the performance needs to be top-tier since it's just going to be solving a Rubik's cube through set/generated algorithms and methods.

3.2 Algorithms Descriptions

3.2.1 Human Algorithm

I felt, for a project like this, it was good to get a feel for the way a program would structured in regards to following set steps and rules in order to solve a Rubik's Cube. I knew the basic methods of solving a cube already which definitely helped when implementing this algorithm. This is a very time efficient method when a computer is doing it but it does require a lot of moves to solve a cube from a scrambled state. I found it takes between 225 - 275 moves on average to solve any given cube.

7 Step Algorithm

This is the most bare-bones, basic algorithm on the internet for solving a Rubik's Cube. I chose this one because it had a larger number of steps to follow, therefore making it easier for me to plan out and implement into my program.

1. **White Cross [ha1]**
   I needed to get the program to locate the white edges and position/orient them according to where the corresponding centers were. E.g If an edge had a white and red stickers,

the edge needs to be placed in between the white and red centers. This is applied to all edges eventually.

2. **White Corners** [**ha2**]
   The corners with white are found and positioned into the appropriate, corresponding centers.

3. **Second Layer** [**ha3**]
   The middle edges on the "E slice" are positioned and oriented according to the center colours on the cube.

4. **Yellow Cross** [**ha4**]
   Make a yellow cross on top of the cube, we don't have to pay attention to the edge colours.

5. **Yellow Edges** [**ha5**]
   Swap the yellow edges so the other colour on the edge is aligned with the appropriate center.

6. **Permute Corners** [**ha6**]
   Position the corners with yellow in the correct places on the cube. The orientation doesn't matter.

7. **Orient Corners** [**ha7**]
   Orient the corners with yellow to solve the cube.

3.2.2 Search Algorithm

My original intentions were to implement a local search algorithm. However, after some research, I discovered it would be easier and more reasonable to implement a generic, heuristic based search algorithm as a Rubik's Cube needs a solution that can solve it.

Layer by layer approach

I had the mindset of solving the cube similar to the human algorithm; layer by layer at a time. This worked great for solving the first layer within 15-25 moves.However, I discovered, after implementing the appropriate heuristics, I would be solving the middle and last layers that inevitably higher quantities of moves which takes a substantially larger amount of time to calculate.

If I were to try and pursue this method of solving, it would require generating all possible permutations of moves up to a length of 12 in order to have a good chance at solving the cube. That's 1,224,880,286,215,950 different permutations for my program to test. This would take years to get through just for a single iteration of solving the cube.

Korf's Algorithm

Thistlethwaite's Algorithm

3.3 Pruning Tables

Pruning tables are a fundamental component that should be used by computing related solving algorithms. The solutions, depending on the algorithm used, are discovered using the appropriate search tree method. There are 18 basic moves[**REFERENCE**] that are considered when solving a 3x3x3 cube. It would literally take over one thousand years for a computer to test every single permutation of moves (which isn't very practicable). Even when breaking the problem of a scrambled cube into sub problems such as Thistlethwaite's Algorithm; without pruning tables, it can take over a day to solve a randomly scrambled cube... If not longer. **PERHAPS TEST AND PROVE THIS**.

Pruning tables are used as a reference for what branches of a search tree are worth checking. A pruning function takes a cube's state as a parameter and puts it through an encoding function which returns the appropriate information to retrieve the corresponding number of moves value from the pruning table. If the value is higher than the depth it's currently looking for a solution in, then skip that branch. This saves substantial amounts of time.

I generated these through a Breadth First Search. For Korf's Algorithm which solves the cube outright, the corners$_{oc}corners_{p}tabletakesthelongesttogenerate$.

3.4 Implementations

3.4.1 How is the Cube being Represented?

I created a 3D and a 2D version of the cube. Both versions represent the same Cube object and I'm hoping to add control for the user to change the cube's state via a clickable 'colour board'. This feature would allow people to input their own scrambled cube states for my program to solve.

I discovered throughout testing my various solving algorithms that I was representing my cube in a very expensive way. During my research, I discovered I could represent my cube in a "Bitboard Structure"[**cuberep**] but since there are certain functions not available through Java, I chose to create an Integer array for each face on the cube. Each array represents a face and stores 8 integers. Each integer represents a sticker of a specific face (this excludes the center sticker as they're set in place).

3.4.2 Threading

During the selection algorithm testing, I saw my GUI freeze up. To deal with this, I implemented threading to run the solving functions which meant that while the program is calculating the appropriate moves, I can move the cube to check its current state and change the program settings.

3.5 Experiments design

3.5.1 Print statements

Using a language that doesn't have the greatest debugging capabilities, I relied on print statements quite heavily for informing me on the necessary variable values during run time. When testing different methods of, for example, cloning cube different types of cube objects and testing algorithms, I would print how long it would take for the program to get through that. E.g. I did an experiment regarding the cloning and testing of algorithms on millions of my 3D Cube Objects compared to the cloning and testing of millions of fast-cube' objects (Objects representing the cube state through arrays of integers). I found that my time to clone cubes and test algorithms halved.

3.6 Summary

Chapter 4

Results and Analysis

4.1 Performance

The solver's performance can be measured using various metrics. It could be measured by time taken to solve the cube, computing power used or the number of moves needed to solve the cube from scrambled to solved states.

The Beginner's Method - Human Algorithm[**ha**] works very well at solving the cube in a short time span because it follows a specific set of predefined steps to solve the cube. It takes a couple seconds for the computer to solve the 3x3x3 cube. However, it averages 200 - 275 moves to solve the Rubik's Cube each time. It's not an efficient algorithm in regards to the number of moves it takes to solve the cube. It's not efficient compared to the various other human algorithms that are available like CFOP[**cfop**]. CFOP is capable of solving the cube between 50-75 moves.

Chapter 5

Discussion

5.1 Significance of the findings

5.2 Limitations

5.3 Summary

Chapter 6

Conclusions and Future Work

6.1 Conclusions

6.2 Future work

6.2.1 Robotics

6.2.2 Larger Cube Solves

6.3 Summary