Thomas Davidson

Comparing Search-Based Methods and a Machine Learning Approach to Solving a Rubik's Cube

Part II Project in Computer Science

Corpus Christi College

May 18, 2018

Proforma

Name:	**Thomas James Davidson**
College:	**Corpus Christi College**
Project Title:	**Comparing Search-Based Methods and a Machine Learning Approach to Solving a Rubik's Cube**
Examination:	**Part II Computer Science Tripos, May 2018**
Word Count:	**12,000**[1]
Project Originator:	Thomas James Davidson
Supervisor:	Dr. Sean Holden
Overseers:	Dr. Richard Mortier

Original Aims of the Project

This project wanted to compare different search-based approaches to solving a Rubik's cube, investigate the potential of using machine learning approaches and conduct a controlled experiment to compare the performance of human participants to a trained neural network in the task of solving semi-scrambled Rubik's cubes.

[1]This word count was computed by `detex tjd45_dissertation1.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Work Completed

The work completed consists of: Java program which can store, manipulate and solve Rubik's cubes as well as loading and evaluating the performance of neural network models trained by a machine learning framework; various MATLAB scripts to plot graphical representations of the generated results; multiple statistical analyses of results carried out by hand and this written document.

Special Difficulties

I experienced some special difficulties over the Christmas holiday's, as I experienced fairly severe side-effects to some prescribed medication (Sertraline). These included nausea and dizziness, and meant I was not able to achieve as much as I would have liked to over this holiday. I was then diagnosed with tonsillitis at the start of Lent term and so again lost more time at this point. I am liaising with my Director of Studies currently as to whether any further action needs to be taken.

Declaration

I, Thomas James Davidson of Corpus Christi College, being a candidate for Part II of the Computer Science Tripos , hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed *Thomas James Davidson*

Date 17.05.2018

Contents

List of Figures

Acknowledgements

I'd like to acknowledge the contributions of Jacob Bradley in spending countless hours trying to help me relearn statistics, and Phoebe Segal and Christine Davidson for making this document infinitely more readable. I'd also like to thank my Director of Studies, David Greaves, for his help, support and understanding, and Charlotte Paterson for her help with LaTeXand probability. Finally I would like to thank my peers and friends who took part in my experiment and put up with my unhealthy obsession with Rubik's cubes.

Chapter 1

Introduction

1.1 Motivation

The motivation for this project was a longstanding interest in Rubik's cubes and approaches to solving them. The majority of research into this field, has been preoccupied with finding 'God's algorithm': the fewest moves required to solve any arbitrarily scrambled cube [4]. I was more interested in examining and comparing different human-friendly (search-based) methods of solving a Rubik's cube, and examining how 'advanced' methods could reduce the length of solutions. Whilst this has been done before [2], it would still be a very valuable academic exercise to explore. Additionally it overlaps with many of my personal interests, from Rubik's cubes, to data representation and processing.

In addition to the comparative aspect of the project, I wanted to develop a machine learning approach to a problem that humans find relatively easy. Most people, when handed a Rubik's cube which had only been turned 3 moves, are able to solve it with relative ease. My goal was to develop a neural network which could achieve the same feat, and to push this performance further, towards 10 moves, where a human would struggle greatly. This is, again, an area which has attracted some work[3]. I wanted to draw inspiration from earlier research and tackle a machine learning project properly for the first time.

1.2 Terminology

Throughout this document I will use terminology to refer to different aspects of Rubik's cubes. Though these terms will generally be explained in context, I shall define my intended meaning:

- *Cube* — This will refer to a standard 3x3x3 Rubik's cube.
- *Cell* — One sticker on a physical cube. There are 54 cells per cube.
- *Cubie* — An individual piece of the cube – a corner, or an edge – with 2 or 3 cells associated with it respectively. There are 20 cubies in a cube (centre cells are not cubies).
- *Face* — A face of the cube, consisting of 9 cells. There are 6 in total, they will normally be referred to by position relative to the viewer.
  - *Front Face (F)* - Face facing the viewer, in the centre of the cross of the cube's net.
  - *Right Face (R)* - Face on the right hand side of the cube as viewed from F, the right arm of the cross in the net.
  - *Left Face (L)* - Face on the left when viewed from F, left arm of the cross.
  - *Up Face (U)* - Face pointing up when viewed from F, topmost face of the net.
  - *Down Face (D)* - Face pointing down when viewed from F, below the F face in the net.
  - *Back Face (B)* - Face opposite F, pointing away from the viewer. The lowest face in the net.
- *Turn* — A single 90° turn of one of the faces, from the point of view of this face facing you. The face is indicated by a letter. An uppercase letter signifies a clockwise (CW) turn and a lower case letter an anticlockwise (ACW) turn. There are 6 other types of turns aside from each of the named faces:
  - *Middle (M)* - The vertical layer sitting between L and R. Turned as if it were L.

  - *Equator (E)* - The horizontal layer between U and D. Turned as if it were D.
  - *Standing (S)* - The vertical layer between F and B. Turned as if it were F.
  - X - Rotate the entire cube as if turning R. D becomes the new F.
  - Y - Rotate the entire cube as if turning U. R becomes the new F.
  - X - Rotate the entire cube as if turning F. L becomes the new U.

- *Algorithm* — Not strictly an algorithm in the computer science terminology; algorithm is used to describe a string representing a list of turns. 'Algorithm' will also sometimes be used to refer to a full approach to solving the cube, this is clear from context.

- *Scramble* — An algorithm applied to a solved cube. An 'X-move scramble' refers to an algorithm of X (integer) moves applied to a solved cube.

- *God's Algorithm* — This refers to the fewest moves required to solve any arbitrarily scrambled cube.

Chapter 2

Preparation

2.1 General

I decided to implement the majority of this project in Java. The choice of language was not crucial as I intended to program everything from scratch; therefore the ease of interaction with pre-existing libraries was not critical. Additionally, the general structure of the approach I took (Figure 2) lent itself to an object-oriented approach. For these reasons, along with my proficiency in Java, I chose it as the primary language for my project. As I elaborate on in the Machine Learning subsection of this chapter, there was little benefit from using another language.

2.1.1 Cube State Storage

A fundamental initial challenge was how I stored a Rubik's cube in memory; I decided that I would create a custom class, but before I began implementation, I considered various design choices and two basic options:

- *Cell-by-Cell* — Storing each individual sticker as an entity (54 Cells)
- *Piece-by-Piece* — Storing each physical piece of the Rubik's cube (26 Cubies)

Whilst storing piece-by-piece may have saved space, all human-friendly approaches for solving stem from analysing cell, not piece, locations. Therefore, though it may have been possible to translate between these, it was too complex to justify the time

it would have required.

To begin designing a memory-efficient, effective and user-friendly approach to cube storage, I first calculated the minimum memory required to uniquely describe the state of all 54 cells of a Rubik's cube:

$$N = \text{Number of Cells}$$

$$C = \text{Number of Colours}$$

$$B = \text{Number of Bits Required}$$

$$B = N \times \lceil \log_2(C) \rceil = 54 \times 3 = 162 \text{ bits}$$

The faces of the cube could be stored in 2D arrays of 3-bit values, taking up a total space of 162 bits per cube-state. However, it is not possible to declare data types which are a certain number of bits wide in Java. I could have chosen to store pairs or lines of cells together as shorts, but this would have required lots of complex bitwise operations when it came to implementing the cube manipulation. I therefore also discounted the idea of storing each face as an integer. Instead, I stored each cell as a byte – the smallest data type that could accommodate the six colours and that I could place into a 2D array. The storage requirements for this were:

$$\begin{aligned} B &= N \times 8 \text{ bits} \\ &= 54 \times 8 \\ &= 432 \text{ bits} \end{aligned}$$

Clearly this is an almost 3-fold increase on the minimum requirements; however, the benefits of doing this, including ease-of-interaction in later aspects of the program, greatly outweighed benefits that could be gained from trying to reduce the storage, especially as none of the search-based methods would require storage of a large number of states.

2.1.2 Cube Interaction

A fundamental aid to explaining the project was having a representation of the cube a user could interact with. I wanted them to be able to turn the faces of the cube

using the keyboard and to receive feedback from this immediately on screen. In the interests of simplicity and to give the user a full view of the cube, I chose a net representation to display the cube's current state. Initially, I would do this textually, and aimed to implement this graphically as an extension.

2.2 Search-Based Approach

The search-based approach was intended to model how a human would solve a Rubik's cube. Generally, they would follow set algorithms to transform the cube from one state to another, moving through different stages of being solved. Most search-based algorithms work on the concept of layers, first solving a face, then the whole of the first layer of the cube (the 8 cubies with cells on the bottom face), before solving the second layer (4 cubies on the equator slice), and finally the top layer. Most search-based algorithms are human-focussed and are consequently comprised of smaller sub-algorithms with memorable moves. Occasionally, quicker algorithms, which are harder to remember or perform manually (often if they use the B face repeatedly), are possible, and, where feasible, I made such optimisations, decreasing the total moves required to solve the cube.

2.2.1 Algorithm Selection

I considered numerous different methods for the search-based approach. I was originally keen to investigate two or three fundamentally different algorithms. In literature[5], there are only three general approaches to solving a Rubik's cube:

- *Layer-by-Layer*
- *Block*
- *Hybrid*

Layer-by-layer methods work by solving a complete face, then ensuring the edges surrounding that face are solved, before solving a second layer, and finally the top layer. Block methods work by first solving a block of cells together at a corner, until you have a 2x2x2 section solved, before extending this to 2x2x3 and so on. Hybrid approaches are a combination of these two approaches, and are often quite

complicated. I initially wanted to implement a layer method called Fridrich and a block method called Petrus. I thought it would be interesting to see how these two approaches differed in the lengths of their solutions of a cube. However, after further reading, including Duberg and Tidesström's project[2], I discovered there was actually minimal difference in the length of solutions using these methods. In fact, the development of different methods was primarily used to allow humans to better conceptualise the solving process, as well as to develop human-friendly sub-algorithms. I therefore decided to focus on two layer-based approaches and investigate how one was an improvement of the other. The methods I chose I will call Beginner-CFOP and Fridrich. As an extension, I wanted to implement the Ortega method, which does not operate like traditional layer-based methods, instead working in from the corners. I wanted to see if this produced significantly different outcomes.

Beginner-CFOP

CFOP stands for: Cross, First-layer, Orient, Permute. Beginner-CFOP is the most common approach for first-time solvers of the cube:

1. Create a cross on one side of the cube such that the 4 edge cells (not corner cells) match the centre cell. Additionally the edge cell in the other side of the cubie should match its corresponding centre cell. There is typically no algorithm provided for this, and it is executed intuitively.

2. Place the corners of the first face using a basic algorithm. The first layer should now be complete.

3. Using two basic algorithms, place the 4 remaining edge cubies to complete the second layer.

4. Using one algorithm, form a cross on the opposite face to the initial face, without disturbing the rest of the cube.

5. Repeat another algorithm until those edge cells not in the upper cross match their corresponding centre cells.

6. Use a further algorithm to permute (move into correct place but not necessarily correct orientation) the 4 unplaced corner cubies.

7. Use a final algorithm to orient the final corner pieces and solve the cube.

Fridrich

Fridrich is seen as an extension of Beginner-CFOP and is used by speed-solvers in competitions.

1. Starts the same way as Beginner-CFOP; forming a cross on one face.
2. Using a combination of small algorithms and intuition, solve the first two layers in one step (F2L), by placing 4 corner-edge pairs into the correct place.
3. Orient the last layer(OLL) - Using 57 different algorithms, orient the final layer of the cube.
4. Permute the last layer(PLL) - Using 21 algorithms, permute the final layer, without disturbing orientation.

Ortega

Ortega is a hybrid approach to solving the cube originally used for 2x2x2 cubes. It works by orienting and permuting corners and then using a combination of algorithms utilising M, E and S moves to fill in the remaining cells. The basic steps are:

1. Orient top corners(OTC) - Achieved mainly by intuition and small algorithmic tricks.
2. Orient bottom corners(OBC) using a combination of 7 different algorithms.
3. Place all the corners correctly using 1, or 2, of 6 possible algorithms.
4. Edges:
   (a) Solve three out of four edges on one face using basic algorithms.
   (b) Solve all four edges on the opposite face using a combination of 5 algorithms.
   (c) Place the final edge on the initial face using one of two algorithms, as well as dealing with a rare edge case.
5. Flip midges (middle edges) so they are correctly oriented but not necessarily permuted, using one algorithm, repeated if necessary.

6. Move midges into correct places by swapping them where appropriate, using four different algorithms.

2.2.2 Algorithm Storage

Every human-based approach is made up of sub-algorithms, which require storing and indexing. There are a total of 18 unique turns which can be performed (including central slices). These could be stored in a custom, 6-bit data structure; however, there are large numbers of different sub-algorithms to encode, the potential for error when inputting, as well as ease of identifying algorithms, means it is more logical to use characters and strings. Characters in Java are stored using 2 bytes, meaning I am sacrificing just more than a 2-fold efficiency in storage. For these basic methods, this will not be a problem.

2.3 Machine Learning Approach

2.3.1 Rationale

I initially considered using a machine learning approach following mathematical analysis which showed that storing every possible state of a cube becomes quickly infeasible. Additionally, I wanted to investigate if a fairly simplistic model, such as a neural network, could outperform humans at a basic task of trying to undo moves on a cube. Gods number[4] refers to a half-turn metric, where one turn is either a 90° or 180° turn of a face. For simplicity, I will refer to this as quarter turns as it makes little difference. Incidentally, 'God's Number' for quarter turns is 26[4]. In the following analysis, I assume optimal design choices have been made for storing states of cells and algorithms, rather than the decisions I made. There are 18 distinct moves, 54 cells on the cube, with 6 different colours possible for each one. Therefore, the total amount of storage required for a scramble of length X is as follows:

$$\text{Length of scramble} = X$$

$$\text{Available moves} = A$$

$$\text{Number of colours} = C$$

$$\text{Number of cells} = N$$

$$\text{Number of bits to store cube states} = B_s$$

$$\text{Number of bits to store solution algorithms} = B_a$$

$$\text{Total number of bits required for an } X \text{ move scramble} = B_x$$

The general formula for the total amount of storage required to store all possible permutations and their corresponding solves is:

$$B_X = B_s + B_a$$
$$B_s = A^X \times N \times \lceil \log_2(C) \rceil$$
$$B_a = A^X \times \lceil \log_2(A) \rceil \times X$$
$$B_X = A^X((N \times \lceil \log_2(C) \rceil) + (\lceil \log_2(A) \rceil \times X))$$

Clearly, in our situation $A = 18, C = 6$ and $N = 54$, so this equation can be simplified to:

$$\begin{aligned} B_X &= 18^X((54 \times \lceil \log_2(6) \rceil) + (\lceil \log_2(18) \rceil \times X)) \\ &= 18^X(162 + 5X)) \end{aligned}$$

Here is the number of bits required to store the corresponding solutions for some selected lengths of scrambles:

$$B_1 = 3006$$
$$B_2 = 55728$$
$$B_3 = 1032264$$
$$B_5 = 353349216$$
$$B_{10} = 7.57 \times 10^{14}$$
$$B_{15} = 1.60 \times 10^{21}$$
$$B_{20} = 3.34 \times 10^{27}$$

For some context, the value of B_5 is about 42 MegaBytes and B_{10} is approximately 86 TeraBytes. Even B_5 is becoming infeasible to store and search quickly and by the time we reach B_{20} the storage required is in the order of ExaBytes. For this reason, a machine learning model would be a good alternative to a brute-force approach.

2.3.2 Framework Selection

When selecting a framework, I had three criteria in mind. I wanted it to be beginner-friendly, as I had not done significant machine learning work before. I also wanted it to be able to run on almost any platform, and for it to integrate easily with Java, as this was the language I would primarily be using. I selected the framework called Weka as it was available in several distributions, including ones which would run on lightweight versions of Linux. As a lot of my training sets were going to run on an old machine running Lubuntu, this was ideal. Weka is also callable from the command line, which was valuable as I later chose to run some of my training sets through virtual machines. Additionally, Weka works with a very simple .arff data format, which would be easy to generate my own files for. It also has the functionality to use other common data formats. This, coupled with the fact that it was easy to integrate with Java, made it the sensible choice.

2.4 Requirements Analysis

1. Write a Rubik's cube simulator.
2. Write software to solve a scrambled Rubik's cube using a search-based method.
    - Implement traditional human-friendly algorithms (CFOP/Fridrich).
    - Implement more advanced human-friendly algorithms (Roux/Petrus/ZZ)
    - *Extension:* Implement advanced, computer-focussed, search-based algorithms (Thistlethwaite/Kociemba).
3. Evaluate the performance of the three approaches in (2).
4. Use mathematical analysis of permutations of a Rubik's cube, to show when it becomes infeasible to store every possible permutation of the cube after X moves.
5. Use a machine learning approach to train a neural network to solve a semi-scrambled Rubik's cube (semi-scrambled: a set number of moves away from being solved).
6. Evaluate performance of (5), as the length of semi-scrambles is increased.

7. Evaluate performance of (2) and (5) on semi-scrambles
8. *Extension:* Introduce machine learning approach to the initial part of the solve.
9. *Extension:* Develop hybrid approach using (8) and (5) to solve a fully-scrambled cube.
    - Contingency for failure: develop hybrid approach using (2) and (5)
10. *Extension:* Evaluate performance of hybrid solution against (2)
11. *Extension:* Build graphical representation of (1)
    - Contingency for failure: Use open-source simulator.

2.5 Software Engineering Best Practice

As this project consisted of software design and analysis, it was important to follow best practice. The first step of this was laying out a clear timeline and plan of approach for the project. This can be seen in my project proposal (Appendix A). To ensure that I observed this plan I generated a Gantt chart to act as an aid to monitor my progress and arranged weekly meetings with my supervisor to ensure that relevant milestones were met. This project differed from traditional software development projects as it was being carried out by an individual. Therefore the issue of version control was less important, but ensuring suitable backups were made of source-code and written documents was essential. To avoid any dataloss I ensured that the source-code and dissertation files were backed up in two separate locations. Both sets were backed up physically onto a hard drive every week and automatically backed up onto GitHub (for source-code) and OneDrive (for dissertation files). Where possible I laid out and explicitly planned the implementation using standard techniques such as UML diagrams.

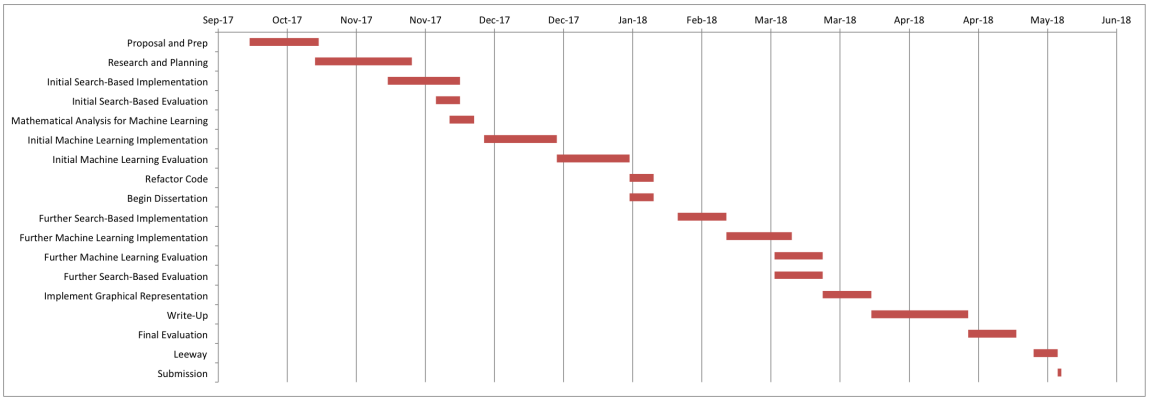

Figure 1: Gantt chart for project.

Chapter 3

Implementation

3.1 Overall Design and Structure

This project would involve the interaction of several different components, the majority Java classes, as well as the integration of a machine learning framework. The primary aim of the software would be to return meaningful, detailed statistics about the performance of the various different approaches implemented. In order to ensure that I went about this in an efficient and clear way I drew up the diagram in Figure 2 to summarise how the final system would fit together. The arrows in the diagram represent the flow of data between components.

3.2 Cube Storage and Interaction

I had investigated different methods of storing the cube and settled on storing each face as a 2D byte array: a suitable payoff between ease-of-use and storage requirements. I implemented this by writing a `Cube` class, with a constructor to create a solved cube. I then implemented methods to facilitate interaction with the cube. Firstly, I programmed the 12 different moves the cube would require (turn of each face clockwise or anticlockwise), as well as the three moves which allow rotation of the entire cube (X,Y and Z); and then I implemented algorithms to perform random and user-defined scrambles.

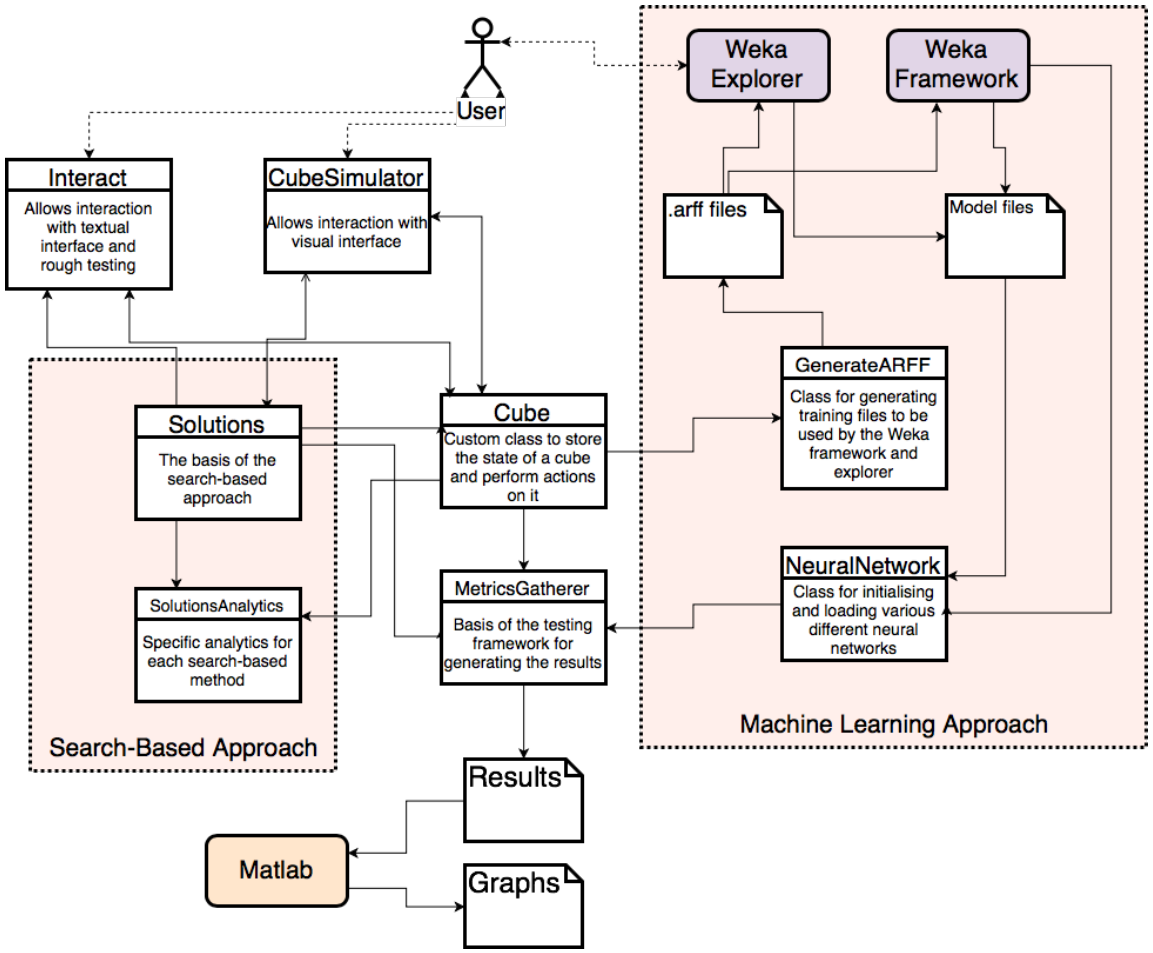

Figure 2: UML-style diagram to show the planned structure of the implementation.

```
383    void performAlgorithm(String alg, boolean p){
384
385        //Split the provided algorithm string into a character array
386        char[] move = alg.toCharArray();
387        //Set the global print variable to the value passed to the
       function (by default global print is true)
388        print = p;
389
390        //loop over the algorithm string
391        for(int i = 0;i<move.length;i++){
392            //this string variable was used when debugging
393            lastMove=move[i];
394
395            //calls the turn function which mutates the cube faces
       depending on the passed character
396            turn(move[i]);
397        }
398
399        //reset print to true
400        print = true;
401    }
```

Listing 3.1: The *Perform Algorithm* function inside the `Cube` class

Next, I implemented algorithms to display the state of the cube. One to display its literal state, in terms of values of the byte arrays, and a minimal textual representation of the cube. The final algorithm I used for displaying the state of the cube printed it out as a 2D net, with each colour assigned a corresponding letter (R,W,B,O,Y,G). I then adapted a second class, called Interact, to allow the user to type in moves and see how this affected the cube.

```
7        //Initialise the cube
8        Cube cube = new Cube();
9        //Display the inital solved state of the cube
10       cube.prettyPrint();
11
12       //Define a new scanner to read from System.in
13       Scanner reader = new Scanner(System.in);
14       //Define a string to hold the current input
15       String input = reader.nextLine();
16
17       //Loop until the user quits
```

```
18        while(!input.equals("q")){
19            //Perform the inputted move on the cube with the print
      variable set to true
20            cube.performAlgorithm(input, true);
21            //get the next move when the user is ready
22            input = reader.nextLine();
23        };
24
25        reader.close();
```

Listing 3.2: The `Interact` class allowing user-interaction with a cube

Sample output of a user performing move:

```
     WWW
     WWW
     WWW
OOO GGG RRR
OOO GGG RRR
OOO GGG RRR
     YYY
     YYY
     YYY
     BBB
     BBB
     BBB

r // <- User Input
     WWB
     WWB
     WWB
OOO GGW RRR
OOO GGW RRR
OOO GGW RRR
     YYG
     YYG
     YYG
     BBY
```

```
    BBY
    BBY

q // <- User Input
```

3.3 Search-Based Approach

3.3.1 CFOP

I began implementation of the search-based solutions to the cube with CFOP. In order to allow different solution approaches to be called from one place, I encapsulated them within the `Solutions` class. The *solve* method of this class would take a scrambled Cube and a string – representing the scramble – as arguments and return another string representing the solution algorithm for that cube. I implemented the 7 steps of the CFOP method separately, before having the overarching *CFOP* method call them. Every step of the CFOP approach requires performing one of a subset of different moves, based on the cube-state; I therefore implemented a function to return the position of edge and corner pieces, based on this indexing system.

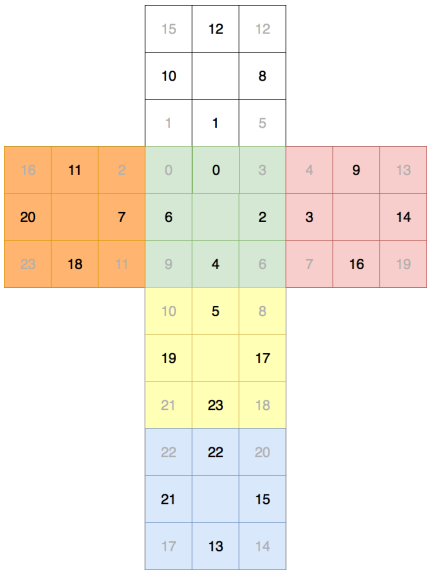

Figure 3: Indexing system where the number represents the first colour passed to the function. The grey numbers represent the indexing system for corner location.

Then, I created an array of the relevant sub-algorithm strings for each stage and indexed them so that the appropriate moves could be selected using the integer returned from the locating function. The first three steps of CFOP all require four pieces to be placed. For each of them I had a section of code that would loop, placing a piece and then rotating the cube, so the same indexing system could be used. This was the basis for the cross, corners and F2L methods. Extra care had to be taken when a target piece lay in the destination of another target piece.

```
1304    static void FBcross(Cube cube){
1305       //initialise loop variable to false
1306       boolean cross = false;
1307       //initialise variable for keeping track of number of pieces
        correctly placed
1308       int limbs = 0;
1309       //global variable for use within analytics
1310       loopcounter1 = 0;
1311
1312       //loop until the cross has been formed
1313       while(!cross){
1314          //check if the target piece has been placed correctly
1315          if((cube.Fface[0][1]==cube.Fface[1][1])&&(cube.Uface[2][1]==
        cube.Uface[1][1])){
1316             //rotate the cube on its axis
1317             cube.performAlgorithm("Z",false);
1318             //record this move
1319             algorithm = algorithm + 'Z';
1320             //increment the number of correctly placed pieces
1321             limbs++;
1322             //check how many pieces are correctly placed and update the
         loop variable
1323             if(limbs == 4){
1324                cross = true;
1325             }
1326          }else{
1327             //get the position of the target edge
1328             int move = getEdgePos(cube,cube.Fface[1][1],cube.Uface
        [1][1]);
1329
1330             //use the returned index to perform the next sub-algorithm
1331             cube.performAlgorithm(crossmoves[move], false);
1332             //record the moves on the global algorithm
```

```
1333            algorithm = algorithm + crossmoves[move];
1334            //update global variables for analytics
1335            counter1[move]++;
1336            loopcounter1++;
1337        }
1338    }
```

Listing 3.3: The main loop of the `FBcross` method to form the cross on the first side of the cube

The top-cross method only required one algorithm but, dependent on the state of the top face, the way, and number of times, it was applied varied; and therefore, I wrote a function to establish the state of the top-face, return this, and use it to determine the next move.

The last two steps followed very similar patterns to the top cross method, and were implemented similarly. In order to verify the expected performance of the overall algorithm, I wrote checker methods to ensure that what was expected at each point had been achieved (for example: all the corners in place), and then ran the `solve` method, with thousands of randomly generated scrambles, and output if any of the stages were not as expected. When outputting it also recorded a trace of the moves it had carried out, allowing me to correct any errors introduced during manual encoding of the algorithms.

```
1341    if((cube.Fface[0][1]==cube.Fface[1][1])&&
1342            (cube.Fface[1][2]==cube.Fface[1][1])&&
1343            (cube.Fface[2][1]==cube.Fface[1][1])&&
1344            (cube.Fface[1][0]==cube.Fface[1][1])&&
1345            (cube.Uface[2][1]==cube.Uface[1][1])&&
1346            (cube.Rface[1][0]==cube.Rface[1][1])&&
1347            (cube.Dface[0][1]==cube.Dface[1][1])&&
1348            (cube.Lface[1][2]==cube.Lface[1][1])){
1349        if(print)
1350            System.out.println("Cross Solved... " + algorithm);
1351    }else{
1352        System.out.println("Error: Cross not solved, algorithm to this
    point: " + algorithm);
1353        cube.prettyPrint();
1354    }
```

Listing 3.4: Checker for `FBcross` method

3.3.2 Fridrich

The Fridrich method takes the seven steps of the CFOP method, streamlines them into four and takes an alternative approach to the final layer. Notably, Fridrich combines steps 2 and 3 of CFOP into a single step: F2L. This means that a corner piece and edge piece are placed at the same time. Therefore, 2 target pieces must be located using the previously implemented locator methods. Once these have been obtained, the correct F2L algorithm needs selecting. As we have 2 indexes, I stored the list of moves in a 2D array. Similar to the early steps of CFOP, this must be repeated 4 times, so the whole cube is rotated after each loop, and the same indexes can be used in each loop.

```
30    protected static String[][] F2lmoves = {{"FrfR","rUURRURRUR","
      URUrUURUr","YUUlULUYLUlYY","uRurURUr","RurUUfuF","uRUrURUr",
31        "yUruRUUrURY","RUr","rFRf"},
32            {"yrUURUruRY","UURRUUruRuRR","RurUUrUURRURRUR","RurfUUF","
      RUUruRUr","yUURRUURUrURRY","UURUrURur","yurUURurURY",
33            "UfuFuUURUrURur","UfuFuufUUFufUF"},
34            {"UfUFufuF","yrURUUYRUr","uRurUURur","uRUrUfuF","URur","
      uRUUrUfuF","uRUUrUURur","yruRY","RurUURUUruRUr","fUFUUyUruRUUrURY"},
35            {"URurufUF","UUfUFURur","","rFRfRurURurUURur","ufUFURur","
      UURurufUF","UufUFURur","RurufUF","fUFURur","uRurufUF"},
36            {"fuFUfuF","uRurURur","RuruRUrUURur","RurUfuFufuF","RurURur","
      UfuFUfuF","UURurURur","ufuFUfuF","URurURur","UUfuFUfuF"},
37            {"fUFufUF","uRUruRUr","RUrUURurURUr","RUruRurUUfuF","RUruRUr",
      "UfUFufUF","UURUruRUr","ufUFufUF","URUruRUr","UUfUFufUF"}};
```

Listing 3.5: 2D array of moves required to perform the F2L stage in the Fridrich approach

The next step of the Fridrich method requires orientation of the top layer. This is done by applying an algorithm, selected based on the state of the top face. I considered various different approaches to uniquely defining the top state of the cube, and devised a method which effectively read the outwards pointing faces of the top face – that is, those that don't point up – and built a 12-bit integer based on the positions of the target colour(top face centre cell). See Figure 4. Then, from the 12-bit integer, I returned an index in the range of the indices of the OLL algorithms, and performed this algorithm. If there was no match, the cube was rotated until there was.

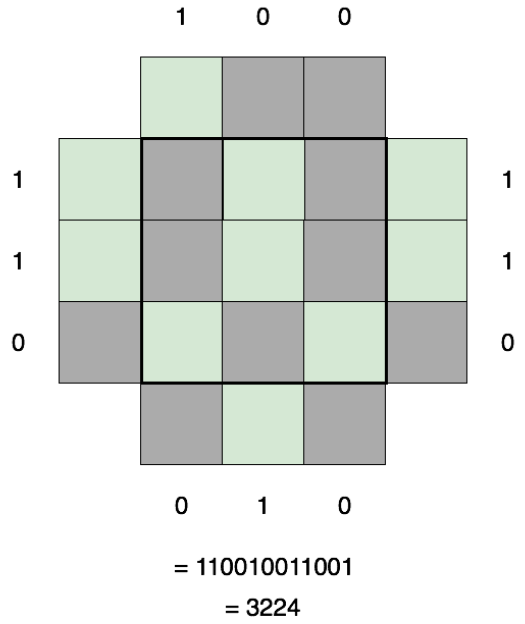

Figure 4: Indexing system to describe state of the top face.

For the final step of the Fridrich method, I employed a similar technique, but the target faces were defined as the ones pointing outwards. Again this algorithm was written to return an index in the range of the PLL algorithms, and then to perform the corresponding algorithm.

Similarly to the CFOP method, I wrote checker methods and performed a large quantity of test cases. Many of the algorithms in the Fridrich method required the use of the M slice move, and so I added this functionality into my `Cube` class.

3.3.3 Ortega

The Ortega method differs significantly from the previous two methods, as it focusses on building from the corners and utilises central slice moves to place the missing edges. Despite this, I managed to successfully re-use aspects of the CFOP and Fridrich approaches. Most useful were the locating functions for edges and corners. I considered using the same algorithm for placing corners, but the CFOP method places the corners explicitly into the correct permutation, and this is not required for Ortega. Instead, I wrote a new function to position the bottom corners. For this,

I made use of an adapted 'get corner' position function and utilised this, with an indexed set of moves, to place the bottom corners into the correct orientation.

Much like Fridrich, Ortega requires a lot of scenario recognition. This is fairly simple for humans, but presents difficulties when trying to approach it algorithmically; as you must either design a way of recognising the state and converting this into an index, or, build unwieldy case statements to recognise scenarios. I built upon the approach I had used for the Fridrich OLL method, and developed a similar way of encoding scenarios in Ortega. Firstly, I used it to encode the number and location of correctly oriented and permuted pairs of edges. Then this 8 bit value was translated into an index referencing the PAC moves string array, and the move applied to the cube, before updating the state of the pairs.

As in steps one and two, for placing the LEdges and REdges (Left Edges and Right Edges), I made use of the 'get edge' positions and translated these into indexes, which were then used to place the target edge into the desired position. Both the *placeLEdges* and *placeREdges* functions were built around loops, which placed one edge at a time, until the desired number was reached.

I then reverted to the method of encoding states into integers, to establish the state of the Midges and locations of those requiring flipping, before again using this as an index.

```
2326    static int getMidgeState(Cube cube){
2327        //initialise state
2328        int state = 0;
2329
2330        //check if a target edge is in place or not
2331        if(cube.Uface[0][1]!=cube.Uface[1][1])
2332            if(cube.Uface[0][1]!=opposite(cube.Uface[1][1]))
2333                //start to build the unique integer by using powers of 2
2334                state = (int) (state + Math.pow(2, 0));
2335        if(cube.Uface[2][1]!=cube.Uface[1][1])
2336            if(cube.Uface[2][1]!=opposite(cube.Uface[1][1]))
2337                state = (int) (state + Math.pow(2, 1));
2338        if(cube.Fface[2][1]!=cube.Fface[1][1])
2339            if(cube.Fface[2][1]!=opposite(cube.Fface[1][1]))
2340                state = (int) (state + Math.pow(2, 2));
2341        if(cube.Dface[2][1]!=cube.Dface[1][1])
2342            if(cube.Dface[2][1]!=opposite(cube.Dface[1][1]))
2343                state = (int) (state + Math.pow(2, 3));
2344
```

```
2345        return state;
2346    }
2347
2348    static int convertMidgeState(int state){
2349
2350        //convert the unique integer to an index
2351        switch(state){
2352        case 0:return 0;
2353        case 3:return 1;
2354        case 6:return 2;
2355        case 12:return 3;
2356        case 9:return 4;
2357        case 15:return 1;
2358        case 5:return 1;
2359
2360        }
2361
2362        //if there is no match then return the index for the move to
        rotate the whole cube
2363        return 4;
2364    }
```

Listing 3.6: Functions to build a unique integer based on the state of pairs, and translate this into an index

I used a very similar approach for the final stage of the solve, involving placing the Midges. I again implemented checker clauses at the end of each step, ensuring the expected results were achieved, or a suitable trace was provided if not.

3.4 Machine Learning Approach

3.4.1 Network Makeup

The Weka framework I selected to implement my machine learning approach worked with the simple .arff file type. The most intuitive feature set was to treat each individual cell as a feature and record the next move as the feature we were trying to predict. Alternatives to this approach included encoding faces or sets of cells as vectors, however, I did not have sufficient time to test multiple different forms of features and, since the early results I gathered were promising, I continued with this

feature set. I wrote a method in my `Cube` class which would generate .arff cases based on multiple scrambles:

1. Generate the .arff file header.
2. Include the 55 attributes (state of 54 cells, and the next turn).
3. Generate random 10-move scramble and record the state of the cube after each move.
4. Invert the scramble by reversing the order and changing lowercase to uppercase and vice versa.
5. Append the state of the cube and corresponding move from the inverted scramble, to the end of the .arff file. (This tells us the next move to perform, based on the state of the cube).
6. Return to step 3 and repeat.

This method was streamlined by ensuring that situations where the cube returned to a solved state were thrown away, so the network was not being trained to make moves when a cube was solved.

Although many sources profess to a rule of thumb for establishing optimal makeups for neural networks, it is widely accepted that there is no real best-method for choosing the makeup of a model beyond trial-and-error. For this reason, I chose a selection of makeups, of varying complexities, and recorded the performance of each of them. I chose the different variations in a structured manner, allowing for easier selection. As Weka does not allow for different layer types, I was limited in what I could vary to the number of layers and the number of nodes per layer. I recorded the performance of these different mixtures, based on the number of correctly classified instances. I also recorded the time taken to train each model, as this was another consideration when assessing tradeoffs. In the table the makeup of the model is defined as the number of nodes in the layer, separated by an underscore if a multi-layer network was used. All the models were trained using Weka's explorer function which enables specification of the details of the model using a command-line or graphical user interface. Most of the tests were run on Microsoft-Azure virtual machines to increase the training I could run concurrently. I elected to use 10-fold cross-fold validation with. I did not use a separate test file and instead partitioned the generated

```
% 1. Title: TenMoves //Header of the file stating information about it
%
% 2. Sources:
%  (a) Creator: Thomas Davidson
%  (b) Date: February, 2018
%
//The name of the relation being trained for
@RELATION nextturn

//List of attributes
@ATTRIBUTE F0 NUMERIC
@ATTRIBUTE F1 NUMERIC
@ATTRIBUTE F2 NUMERIC
.
.
.
@ATTRIBUTE B7 NUMERIC
@ATTRIBUTE B8 NUMERIC
@ATTRIBUTE nextmove {F,f,R,r,L,l,U,u,D,d,B,b}  //Target attribute, listed last

@DATA
//All the instances, with the values of the attributes in order.
1,1,1,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3,5,5,5,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,4,4,4,4,
4,4,4,4,4,5,5,5,5,5,5,3,3,3,U
1,1,1,2,2,2,2,2,2,2,2,2,0,3,3,0,3,3,0,4,5,5,4,1,1,4,1,1,1,1,5,0,0,0,0,0,0,4,4,4,4,
4,4,3,3,2,5,5,3,5,5,3,5,5,3,B
1,1,4,2,2,4,2,2,2,3,3,2,3,3,2,0,0,0,4,5,5,4,1,1,4,1,1,1,1,1,0,0,2,0,0,2,4,4,3,4,
4,3,3,3,3,5,5,5,5,5,0,5,5,0,r
.
.
.
```

Figure 5: Example of generated .arff file

.arff file into a training set and test set using Weka. This was an appropriate way to approach the training as the .arff file was randomly generated and generating a new test file would have achieved the same results.

Model Makeup	Correctly Classified Instances (%)	Model Build Time (s)
20	49.9	24,622
40	54.7	30,167
80	58.6	52,630
160	62.0	74,093
10_10	44.0	12,362
10_20	43.9	14,656
10_40	45.0	18,984
20_10	49.3	16,222
20_20	49.7	22,527
20_40	51.0	34,172
40_10	53.4	37,038
40_20	54.7	43,709
40_40	55.8	54,202

3.4.2 Finetuning

For the initial results I took the percentage of correctly classified instances to be a suitable performance measure. Later on I explain why this is not necessarily the best performance measure and evaluate the performance more thoroughly, coming to the same conclusions. The results show that the 160-node single-layer model performed best. Whilst the multi-layer models were beginning to show some improvement they took a prohibitively long time to train and so I pressed on with a single-layer network. I attempted to finetune this network utilising Weka's in built serialisation, but even after extended discussions with developers of the software I was not able to satisfactorily load and train models further. The original plan had been to alter the learning-rate and momentum following another detailed test plan. As a result, I instead used the trained 160-node model and looked at alternative ways of improving the performance of the network without having to retrain (discussed in evaluation).

3.5 Testing Frameworks

Both the search-based method and the machine learning approach were subject to continuous testing and optimisation during development. When I was satisfied with the initial results for these, I wrote more formal testing classes, in order to gather in-depth data about the solution approaches.

For the search-based testing, this consisted of a large method which, taking the solution method and number of iterations as input, and writing the results to a .csv file. Based on a large loop, it generated random scrambles a specified number of times, from one-move scrambles up to twenty-move scrambles (for reasons discussed in section 2.3.1). Each generated scramble was applied to a solved cube, then this scrambled cube was passed to the chosen solution method and the length of the solution was recorded. This was then written to two separate .csv files: the raw data and an overview file. In addition to this, it also calculated the average solution length at each scramble length.

I chose to have the method loop 1,000,000 times at each scramble length. The rationale behind this was achieved by running further tests which repeatedly ran iterations until there was no change to the average solution length to 3 decimal places for 1,000 consecutive iterations and then terminated, returning the number of iterations. All the scramble lengths returned values between 100,000 and 800,000 so I chose a suitable upper limit.

I then wrote a method which recorded the same information about average lengths for each of the separate steps, allowing me to see how the average length of a solve was distributed across the different steps of the solution approach.

For the neural network, I implemented a different method, which incremented the number of moves in a generated scramble, and recorded the number of cubes successfully solved out of the sample (again 1,000,000). These results were written to a .csv. I also wrote an extension class allowing me to analyse the performance of the network in predicting the first, second, third move etc. in an X-move scramble, and wrote this to another .csv.

I finished my testing framework by writing various MATLAB scripts to allow me to plot these results graphically and analyse the results.

```
bprint = true;
nummoves = 20;
figuretitle = strcat('Scram',int2str(nummoves),'Overlay');
figure('Name',figuretitle,'NumberTitle','off');
hold on
produceOverlayFig(20,'Ortega',true,true,false);
produceOverlayFig(20,'CFOP',true,true,false);
produceOverlayFig(20,'Fridrich',true,true,false);

legend('Ortega','CFOP','Fridrich');

fileloc = strcat('/Users/ThomasDavidson/Documents/Cambridge/Part II
    Project/Dissertation/figs/',figuretitle);

if bprint
    print(fileloc,'-depsc')
end
```

Listing 3.7: MatLAB script to plot figure 44

3.6 Algorithm Pruning

During development of the search-based solutions I wrote several pruning algorithms to remove non-turning moves from algorithms (such as X, Y and Z). Upon inspection of various solution algorithms, I realised the concatenation of different steps was leading to unnecessarily verbose solutions. For example, if one step ended with a U move and the next began with a u move this was two wasted moves. To counter this, I wrote a more aggressive pruner which stripped out the non-turning moves, as well as deleting counteracting moves; for example – "Uu"→"". It also replaced triple turns with single turns, – "UUU"→"u" – and deleted (rare) quadruple turns. The result of this meant I successfully reduced the length of solutions by around 13% on full solution algorithms.

3.7 Graphical Representation

One of my extensions was to develop a graphical representation of the cube. I achieved this using the SWT package[1] for Java. I implemented the depiction of the cube as a simple 2D coloured net, assigning each face a different colour. Next, I linked up the representation to the interaction algorithms I had, enabling scrambling of this visual representation using the keyboard. Finally, I linked up the solution approaches so that the visual representation would execute the solution algorithm for a cube using the selected method, delaying 100ms between moves so it was visible.

```
156                 //split the solution algorithm into individual moves
157                 final String[] moves = solution.split("(?!^)");
158                 //initialise counter
159                 counter = 0;
160 
161 
162                 //loop to update the display every 100ms
163                 Display.getDefault().timerExec(100, new Runnable() {
164                     @Override
165                     public void run(){
166                         //checks that the number of moves performed is
    less than the length of the algorithm
167                         if(counter<moves.length){
168                             //performs a move on the cube
169                             cube.performAlgorithm(moves[counter], false);
170                         }
171                         //updates the screen
172                         canvas.redraw();
173 
174                         if(counter!=moves.length){
175                             counter++;
176                             Display.getDefault().timerExec(100, this);
177                         }
178 
179                     }
```

Listing 3.8: The display update loop in the `CubeSimulator` class

This proved a very valuable resource for a variety of reasons. It allowed me to explain and demonstrate my project and the outcomes of it to people, as well as

[1] https://www.eclipse.org/articles/Article-SWT-graphics/SWT_graphics.html

allowing me to visualise different situations easily. It proved particularly useful when observing how the neural solve approach would fail to solve a scramble (discussed in evaluation).

I did consider developing a three-dimensional representation of the cube, however, I concluded this would not be as valuable a resource, as you would not be able to see the state of every face of the cube simultaneously. I therefore decided it would not be an efficient use of time to develop this.

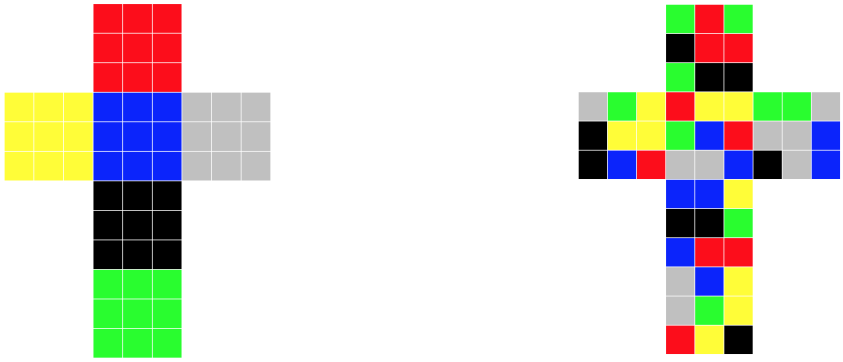

Figure 6: Solved and scrambled cube generated using the `CubeSimulator` class

Chapter 4

Evaluation

4.1 Search-Based Approach

In the following evaluation the graphs represent empirical recordings of performance over a set, high number of iterations. As the results were being used for comparison – and I test significance – and not to explicitly state performance measures I have not included confidence intervals.

4.1.1 Beginner-CFOP

On initial graphs of the performance of CFOP, in terms of solution length for an X-move scramble, I noticed unusual artefacts appearing in the graphs. For odd-lengths of scrambles the results were as expected: a distribution with a peak around the average (Figure 7. However, when an even-length scramble was used, CFOP would only return solutions with an even length – Figure 8.

After thorough investigation I pinpointed the cause of this peculiarity. Initially, I believed that the problem could be related to the pruning algorithm. In order to test this hypothesis, I wrote a new test which returned unpruned algorithms and plotted the lengths of these instead.

As you can see from Figure 9 and Figure 10, the same artefacts appear again. After extensive testing, I could not establish any cases where the `prune` algorithm returned an algorithm as even when originally it was odd. I discounted this theory.

I then investigated the possibility of the moves selected at each of the seven steps of the solution being the cause. To check this, I wrote a function to record the

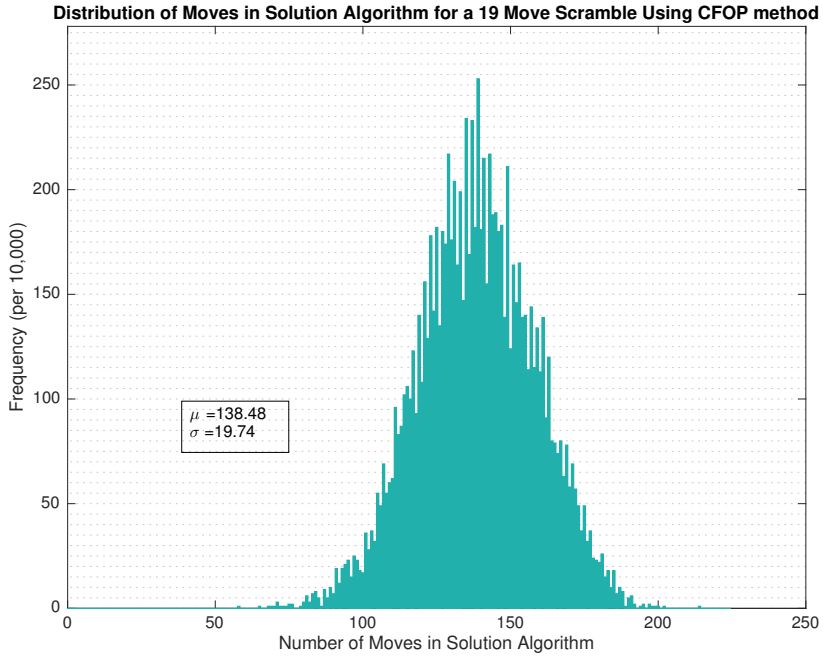

Figure 7: 19-Move Scramble CFOP Pruned-Solution Distribution

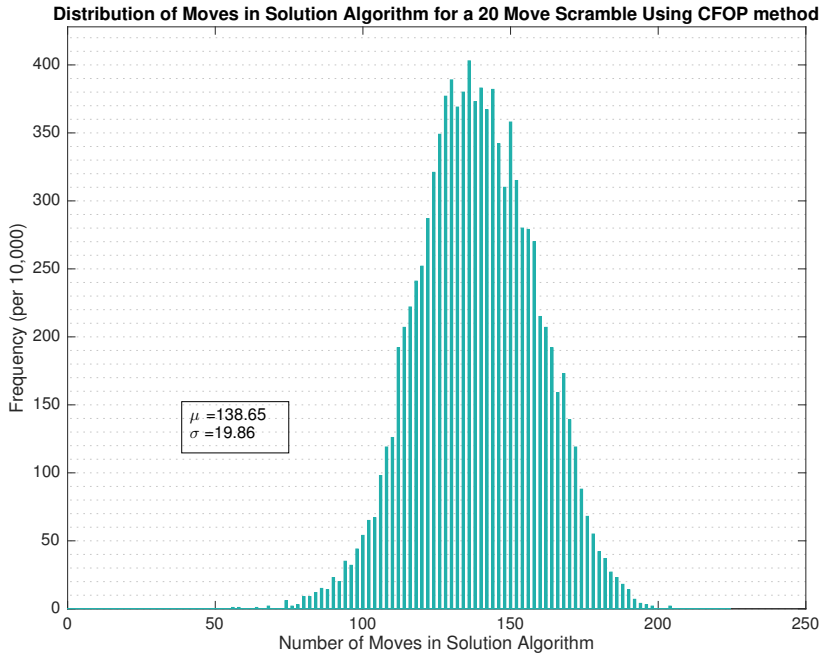

Figure 8: 20-Move Scramble CFOP Pruned-Solution Distribution

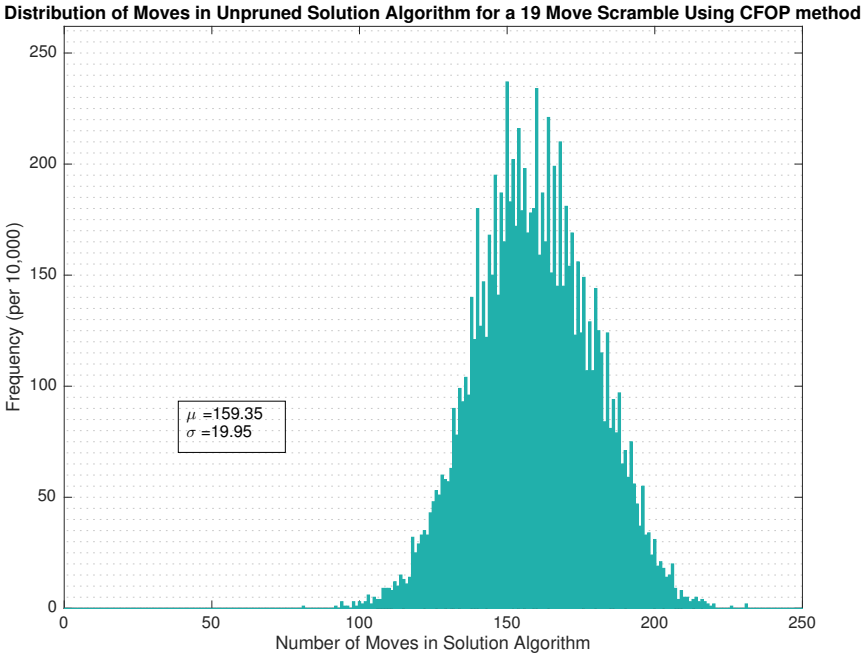

Figure 9: 19-Move Scramble CFOP Unpruned-Solution Distribution

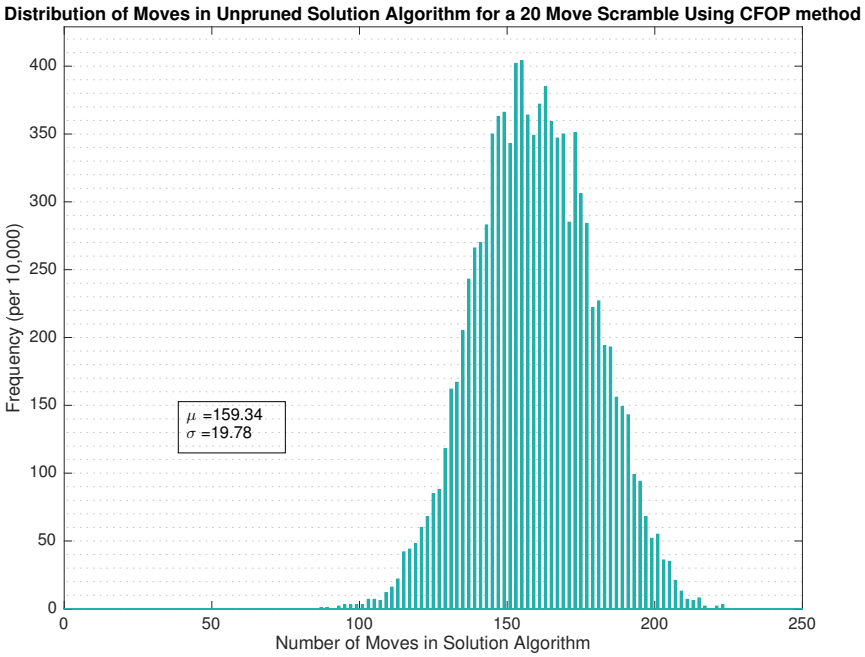

Figure 10: 20-Move Scramble CFOP-Unpruned Solution Distribution

lengths of each step and the number of odd and even sub-algorithms at each step, and iterated this over 10,000 scrambles for 19 and 20 moves[1].

	19-Move Scramble		20-Move Scramble	
Step	Number Odd	Number Even	Number Odd	Number Even
1	4906	5094	4963	5037
2	4989	5011	5025	4975
3	4961	5039	4986	5014
4	4983	5017	5051	4949
5	4937	5063	5051	4949
6	4983	5017	5051	4949
7	4937	5063	5051	4949
Combined	5476	4524	0	10,000

Again, there was no obvious disparity between the results.

I then looked at whether even-length scrambles and odd-length scrambles were leading to a different distribution of move-selection. What I mean by this is that, at each step in the solution there are a limited number of moves available, and so, I wanted to see if even-length scrambles were picking sub-algorithms with only certain indices, which then happened to add up to even numbers, whereas odd scrambles were resulting in a wider variety being selected. In order to test this, I indexed the sub-algorithms at each step and plotted the total number of times they were selected and the number of times each step looped, over a long test. I chose to check the number of times each step looped to see if this too could be causing the return of even-length solutions. Again, there was no discernible difference between the results.

[1]These results were replicated across all odd and even scrambles.

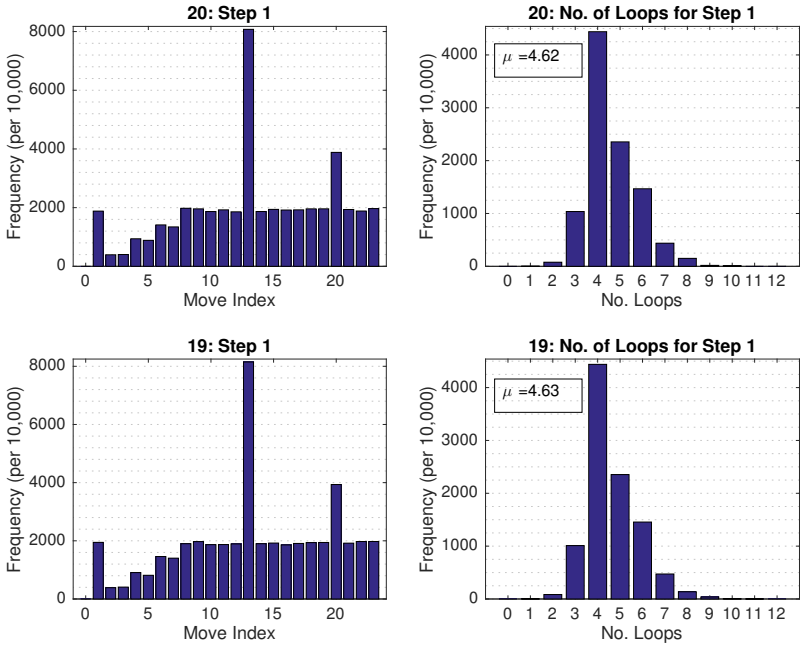

Figure 11: Step 1 CFOP Move Selection

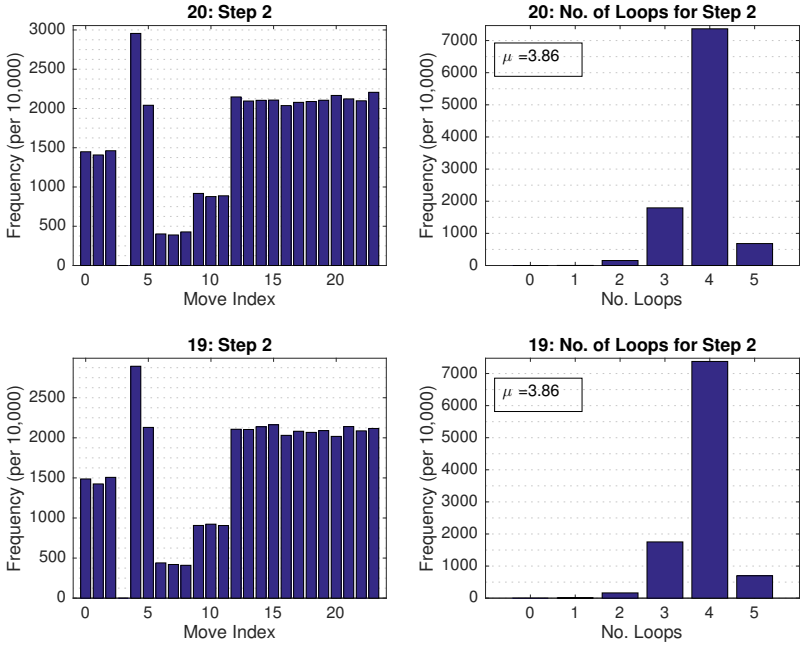

Figure 12: Step 2 CFOP Move Selection

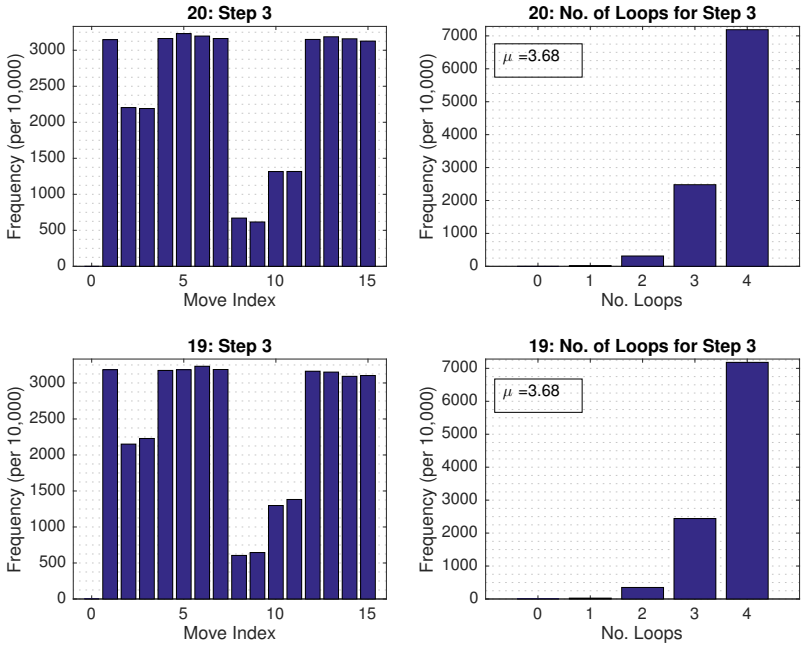

Figure 13: Step 3 CFOP Move Selection

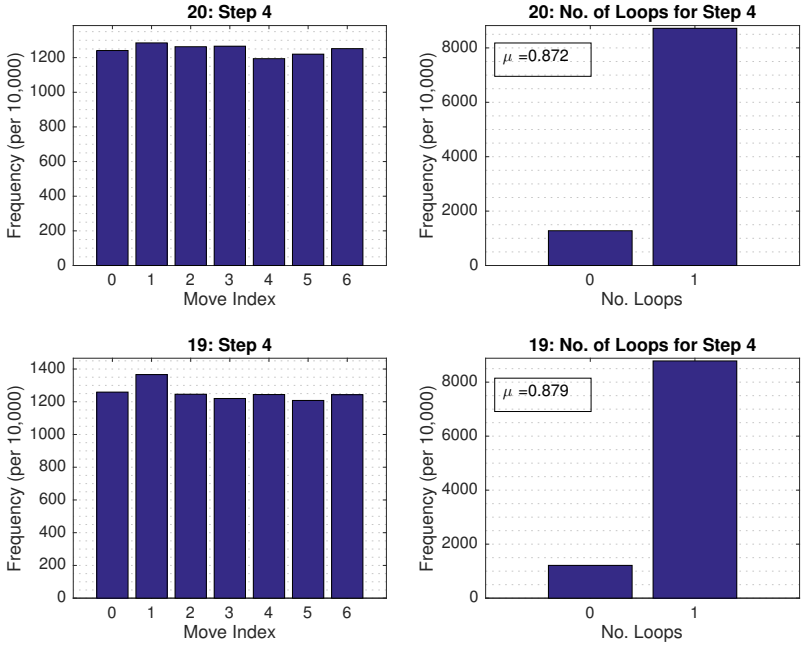

Figure 14: Step 4 CFOP Move Selection

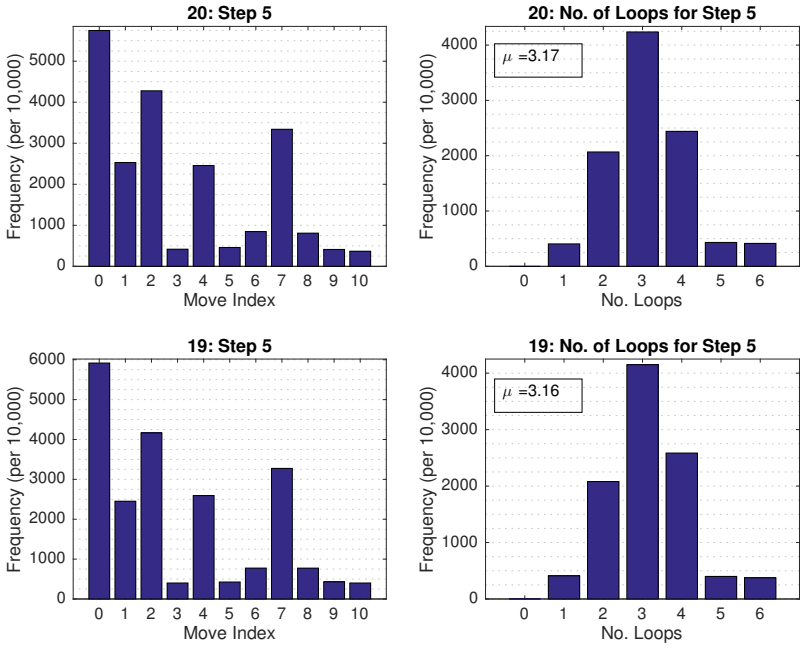

Figure 15: Step 5 CFOP Move Selection

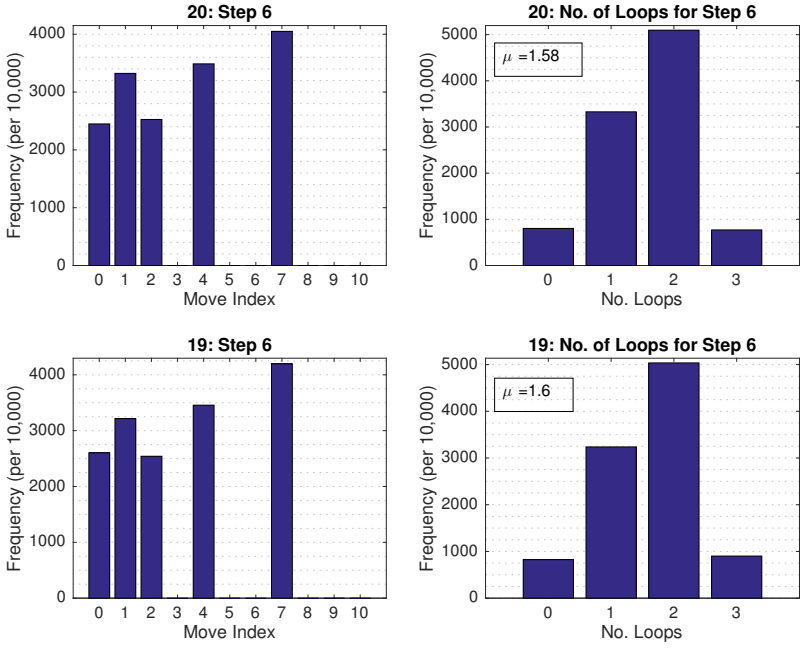

Figure 16: Step 6 CFOP Move Selection

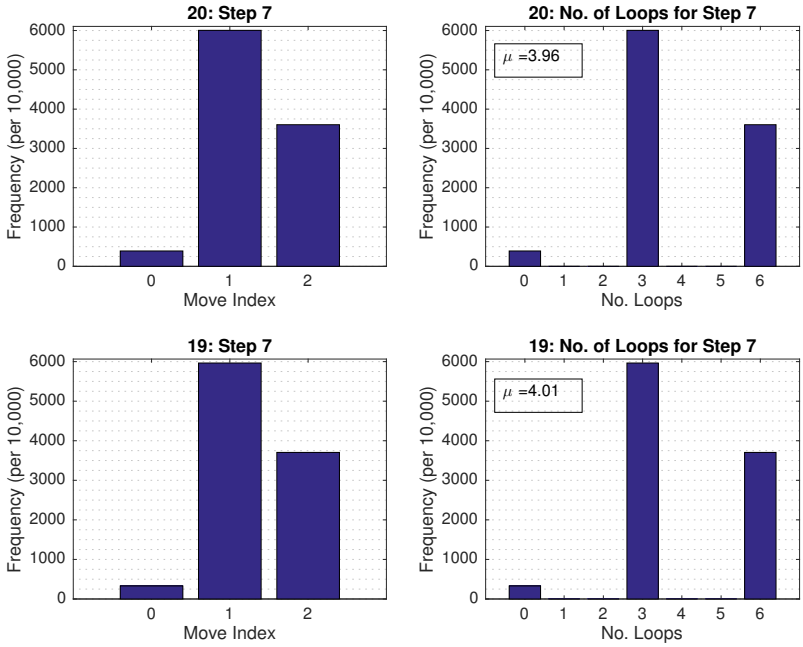

Figure 17: Step 7 CFOP Move Selection

Having failed to establish an obvious explanation for these results, I conducted further research to ascertain the cause. Erik Demaine and his colleagues from MIT produced this paper[1] investigating the mathematics and group theory behind Rubik's cubes, which provided the answer. Most of the mathematics in the paper was beyond the scope of this project, but they showed that if a Rubik's cube was turned an even number of times it would require an even number of moves to solve. Therefore, by extension, an odd-scrambled cube should require an odd number of moves to solve. I realised that the irregularity in my data was not that the even-scrambles were resulting in even-solves but that the odd-scrambles were resulting in a mixture of odd- and even-solves.

After significant further testing, I established that the root cause was in my *scramble* method. In this excerpt, you can see that a random string is generated, one character at a time, and then checked to see if it is a valid scramble by pruning and comparing its length to the original.

```
229    //Original function to scramble a cube using a set number of turns,
       later found to be flawed
230    String flawedScramble(int X,boolean p){
```

```
231         if(!p){
232             print = false;
233         }
234         char[] moves = {'F','f','U','u','R','r','L','l','D','d','B','b'};
235         Random rn = new Random();
236 
237         boolean valid = false;
238         String scramble = "";
239         String checkvalid = "";
240         while(!valid){
241 
242             scramble = "";
243 
244             for(int i = 0;i<X;i++){
245                 char move = moves[rn.nextInt(12)];
246                 this.turn(move);
247                 scramble += move;
248             }
249 
250             checkvalid = Solutions.prune(scramble);
251             if(checkvalid.length()==scramble.length()){
252                 valid = true;
253             }
254         }
255         if(p)
256             System.out.println("Cube scrambled with a "+X+" move scramble:
    "+scramble);
257         print = true;
258         return scramble;
259     }
```

Listing 4.1: The original flawed `scramble` method

This is done to establish whether moves are being done and then undone. However, the move was being performed on the cube immediately after the character was generated, and not after the scramble had been verified. This meant that, if the scramble was invalid, extra moves had been performed on the cube that were not registered. For example, if the cube was initially scrambled with the algorithm "RUu", this would be rejected as an invalid algorithm and pruned to "R". It would then be scrambled with another generated algorithm. If this second algorithm, for example: "LFU", passed, then (still assuming 3-move scrambles) the function re-

turned the 3-move scramble as the scramble performed; in reality, however, a 6-move scramble ("RUuLFU"), which is really a 4-move scramble ("RLFU"), had actually been applied to the cube. In other words, the analytics I was running thought it had applied an odd-scramble and the resulting even-length solution was attributed to that, when, in reality, it had applied an even-scramble, and the even-solution was not erroneous at all. Whilst the cause of the error was due to an incorrectly implemented function, it was very hard to detect for two reasons. Firstly, if an even scramble was rejected – and consequently longer than intended – it would still be even, regardless of iterations. The second reason for the difficulty in detecting was due to the probability of a scramble being rejected:

$$\mathbb{P}(\text{Scramble being rejected}) = \mathbb{P}(\text{Move and its inverse appear consecutively in the scramble})$$

$$\mathbb{P}(R_N) = \mathbb{P}(\text{Scramble of length N is rejected})$$

$$\mathbb{P}(R_2) = \mathbb{P}(\text{Any move is chosen}) \times \mathbb{P}(\text{The move's inverse is chosen})$$

$$= \left(\frac{12}{12} \times \frac{1}{12}\right) = \frac{1}{12}$$

$$\mathbb{P}(R_3) = \mathbb{P}(\text{Rejected in first 2 moves}) + \mathbb{P}(\text{Rejected in 2nd and 3rd move})$$

$$= \mathbb{P}(R_2) \times 1 + \left(\frac{11}{12} \times \left(\frac{12}{12} \times \frac{1}{12}\right)\right)$$

This generalises to:

$$\mathbb{P}(R_N) = \sum_{i=0}^{N-2} \left(\frac{1}{12} \times \left(\frac{11}{12}\right)^i\right) = 1 - \left(\frac{12}{11}\right)^{1-N}$$

This is the probability of an N-length scramble being rejected. We are looking for the probability that an odd scramble results in an odd number of moves being applied. If an odd scramble is intended, but it is rejected on the first attempt and accepted on the second, then this results in an even number of moves being applied. In order for it to actually be an odd number of moves, we are looking for the probability of the scramble being accepted at the first, third, fifth etc. pass:

$\mathbb{P}(O_N)$ = Probability of a odd number of moves for a scramble of length N (N is odd)

$$= \mathbb{P}(\text{Scramble being accepted on first, third, fifth time ....})$$

$$= \mathbb{P}(\text{Scramble accepted on first}) +$$

$\mathbb{P}(\text{Scramble rejected on first}) \times \mathbb{P}(\text{Scramble rejected on second}) \times \mathbb{P}(\text{Scramble accepted on third}) ...$

$$= (1 - \mathbb{P}(R_N)) + (\mathbb{P}(R_N) \times \mathbb{P}(R_N) \times (1 - \mathbb{P}(R_N))) + ...$$

$$= \sum_{i=0}^{\infty} \left(\mathbb{P}(R_N)^{2i} \times (1 - \mathbb{P}(R_N))\right)$$

$$= (1 - \mathbb{P}(R_N)) \times \sum_{i=0}^{\infty} \mathbb{P}(R_N)^{2i}$$

$$= (1 - \mathbb{P}(R_N)) \times \frac{1}{1 - \mathbb{P}(R_N)^2}$$

$$= \frac{\left(\frac{12}{11}\right)^{1-N}}{1 - \left(1 - \left(\frac{12}{11}\right)^{1-N}\right)^2}$$

If we look at this value for a scramble of length 19, we see that:

$$\mathbb{P}(O_{19}) = 0.558$$

In other words there was a 46% chance that a scramble of length 19 would result in an even-scramble, and this explains the graphs. This was compounded by the graphs looking erroneous for the even case when the erroneous cases were in the odd-scrambles. Whilst it was a time-consuming exercise to pinpoint the cause of the error, it did leave me with confidence that my approach was successfully solving the cube each time, as well as with several useful analytic functions which I utilised throughout this evaluation. Solving this problem also had the by-product of reducing the average number of moves to solve the lower-move scrambles (as before, some solution lengths were being included which were actually the result of solving longer scrambles).

It is not surprising that as the number of moves increases, so does the number of moves required to solve the cube. However, as explained in the earlier stages of the

dissertation, there does come a point after which scrambling the cube further does not actually take it to a 'more scrambled' state. This number of moves is 'God's number'[4], and that is why most of the analysis in this document will use 20 moves as a 'fully-scrambled cube'. To show that this is indeed the case, – Figure 18 shows the average number of moves required to solve a cube using the CFOP method, for increasing lengths of scrambles. It was generated by running 1,000,000 scrambles of each length, from 1 to 40, and comparing the lengths of the solution algorithms.

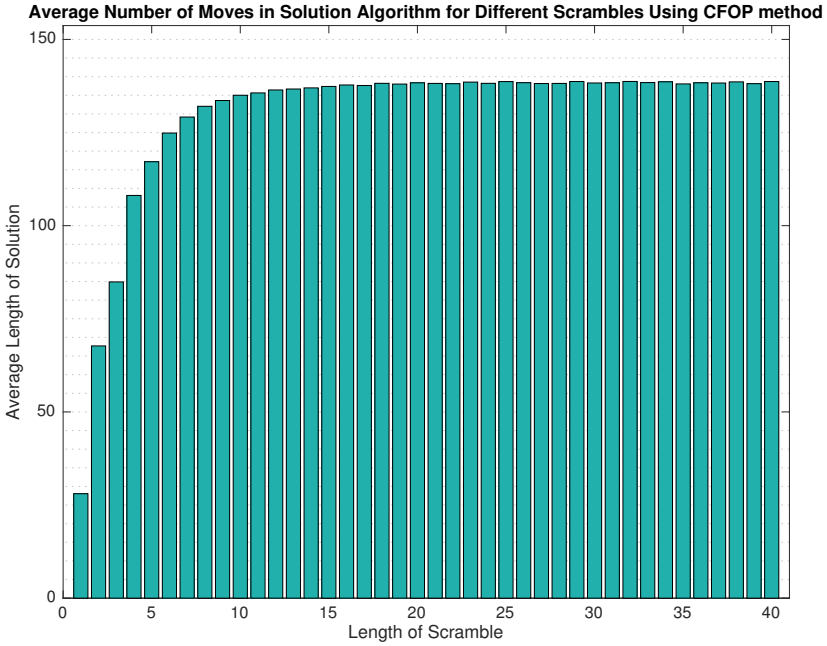

Figure 18: Average CFOP Solution Lengths

The average solution length levels out after about a 12-move scramble. This is due to the nature of the implementation of this solution approach, where algorithms are chosen from a subset, and is explained in more depth shortly.

As the number of moves in the scramble increases, we start to see the peak of our graph, as well as its general shape, move towards the average solve length of around 138. The lower scramble numbers, such as 2 ,3 and 4 moves, all have a significant proportion of their solutions effectively undoing the scramble by chance. As the length of the scramble increases, the entropy and disorder of the cube mean that this becomes decreasingly likely. As the algorithms involved in this approach

are not designed to solve large swathes of the cube in one fell swoop, but instead to solve sections of the cube whilst leaving the rest of it intact, solving incrementally, the mean number of moves to solve increases. As the lower move scrambles have lower levels of disorder and often large sections of the cube still intact, the cube will frequently be solved without requiring further steps. This results in the average being dragged significantly down and also to the standard deviation for the lower move scrambled cubes solutions being higher.

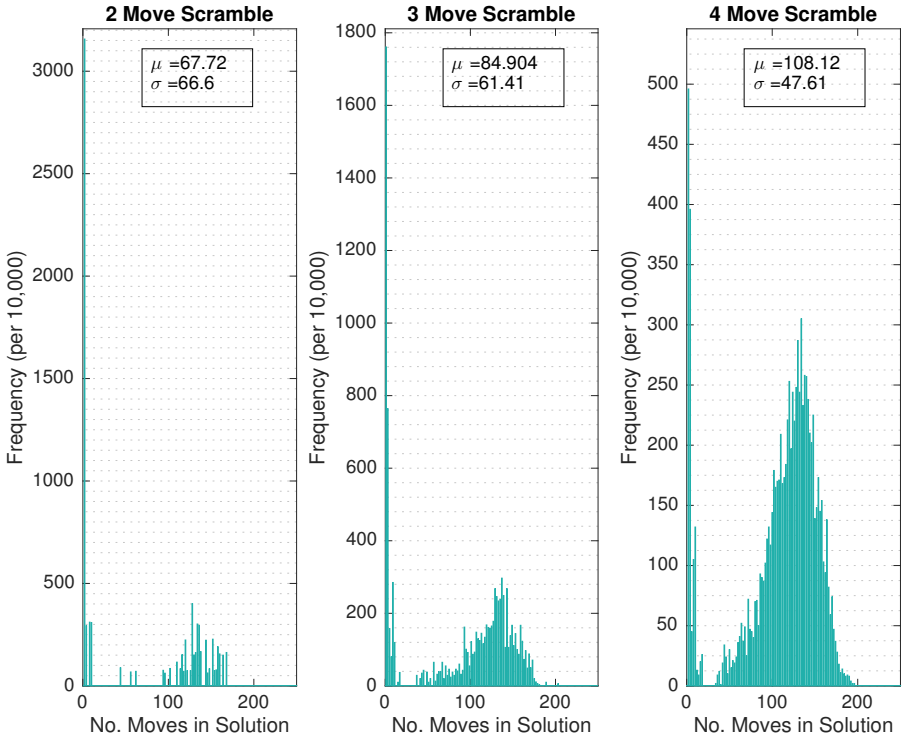

Figure 19: Short-Scrambles CFOP Solution Distributions

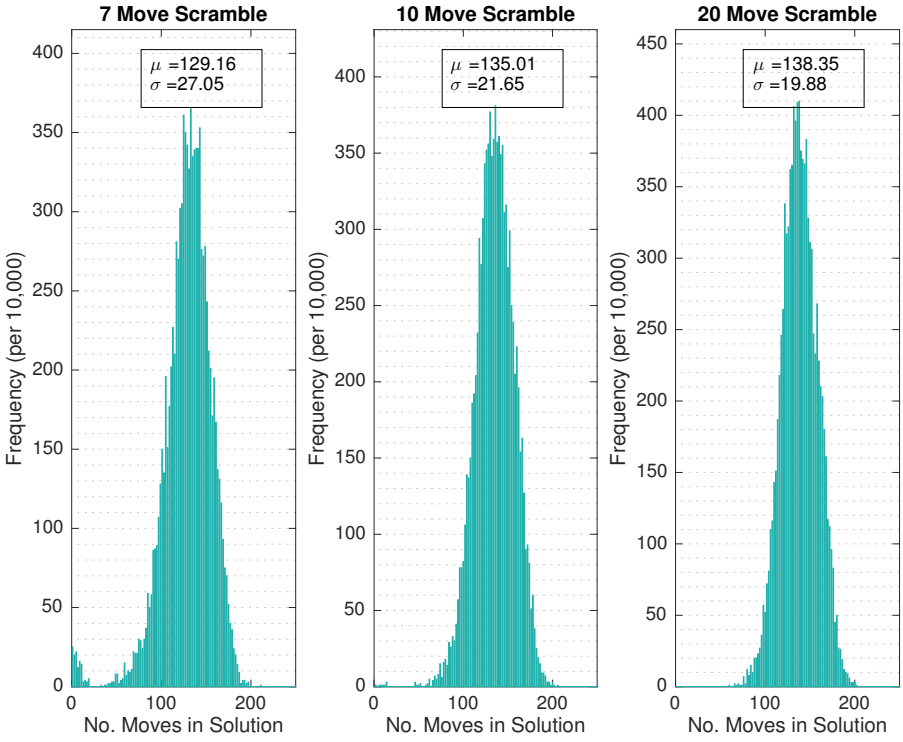

Figure 20: Long-Scrambles CFOP Solution Distributions

Why the plateau and why so early? Each of the 7 steps of CFOP has a limited number of possible actions. The distribution of the lengths of the sub algorithms within each step is shown below. I have omitted the graph for step 7, as it consists of the same 8-move algorithm being used.

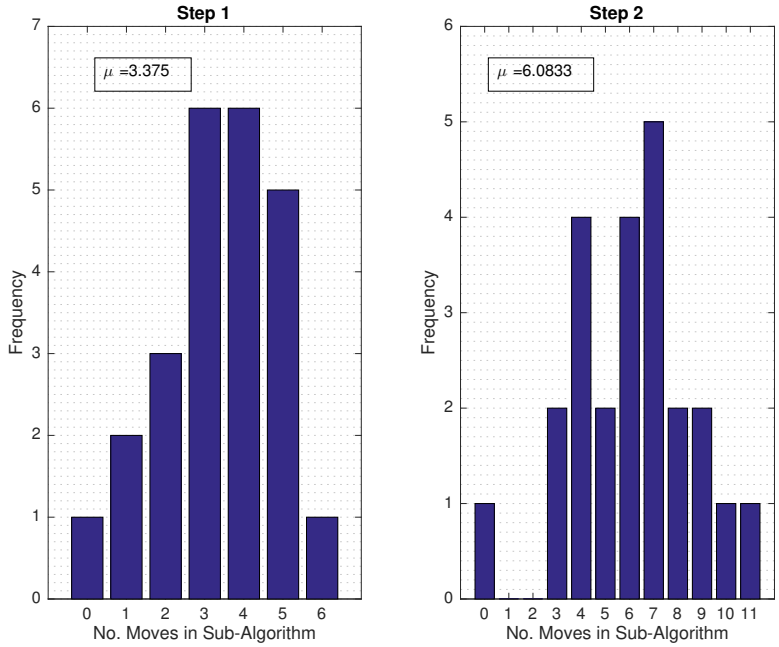

Figure 21: CFOP Step 1 and 2 Move Selection Distribution

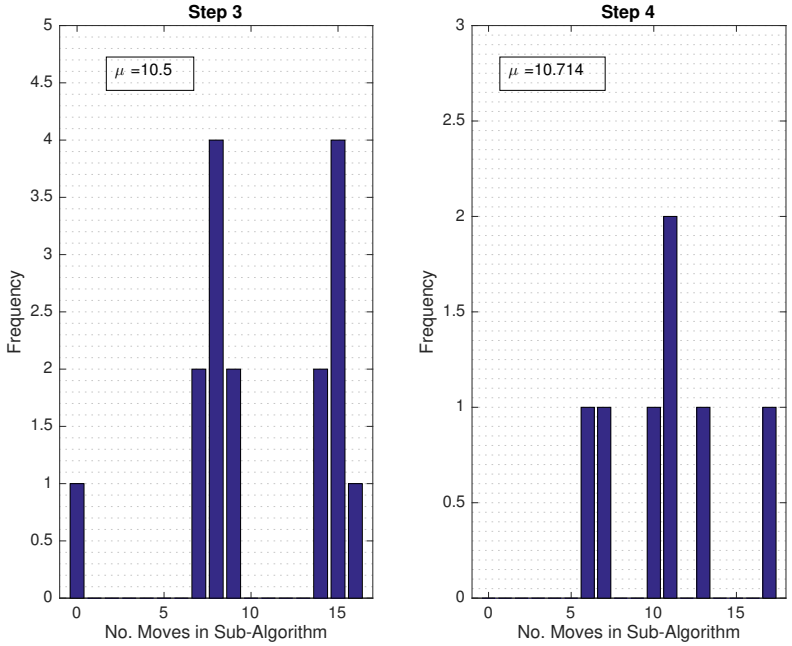

Figure 22: CFOP Step 3 and 4 Move Selection Distribution

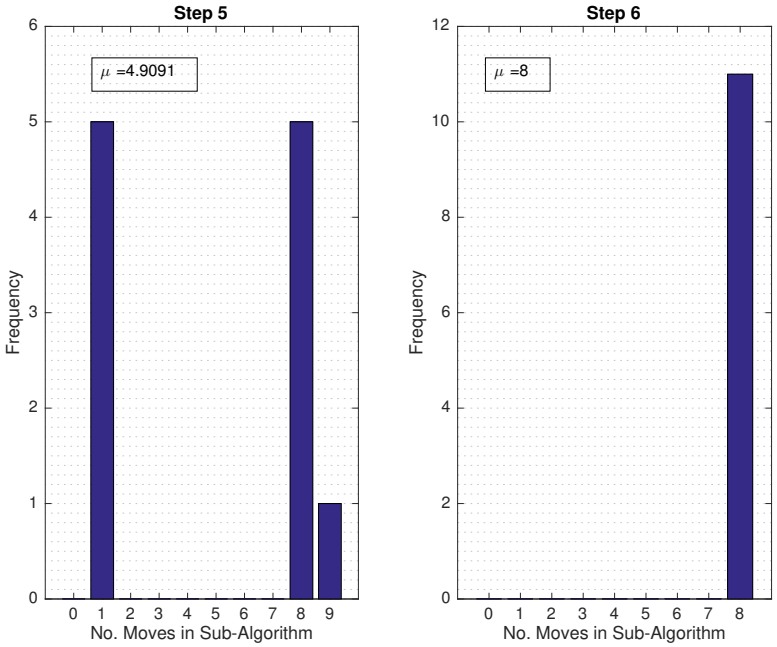

Figure 23: CFOP Step 5 and 6 Move Selection Distribution

As every solution algorithm is a linear combination of these moves, combined with the number of times each step might loop, then if every step is required, there is a limited range of possible lengths. From a minimum (when every step is used) of 23 to a maximum of 310[2]. Moreover, if we take the average length of each sub-algorithm in each step, as well as the average number of loops, and combine these, we find the length of an arbitrarily selected solution algorithm to be 147 moves. This is not the same as the actual average, though it is close. The reason for this is that each sub-algorithm is not equiprobable. You can see this in the graphs showing the frequency of different moves' selectionA. Explaining why this is the case is beyond the scope of this project.

[2]The reason these numbers may not appear consistent with the graphs is that each of the steps has the potential to loop. See the previous figures to see the potential number of loops per step

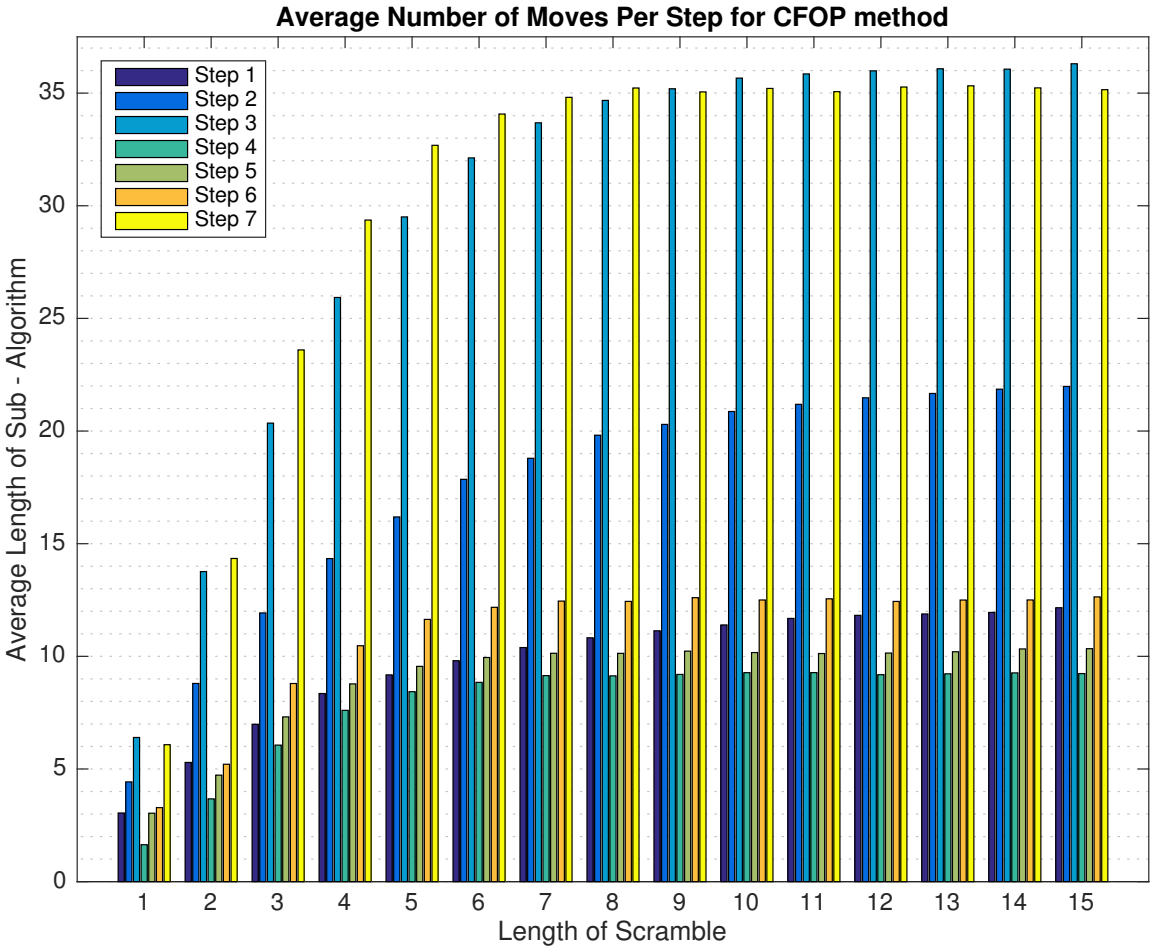

Figure 24: Per-Step Averages for CFOP

This graph shows that as the length of the scramble increases, the number of length of each step begins to tend towards the step's average length, and there are fewer instances of steps being missed. This is why the plateau of the average number of moves comes early.

4.1.2 Fridrich

At early stages of evaluation, I saw the same discrepancy that I had seen with CFOP. However, on this occasion there were still a low number of odd-length solutions for even-scrambles even after applying the solution to the problem outlined above. I es-

tablished the cause of this was that the Fridrich method makes use of M moves, where the middle slice is rotated. My solution algorithm was counting this as one move, when in reality it can be interpreted as two moves. I demonstrated this by adding a clause in the *prune* method to replace M with appropriate moves: "M"→"Rl" or "m"→"rL". However, I decided not to remove these, as the Ortega method is built up of M slice moves, so I treated this as a normal move.

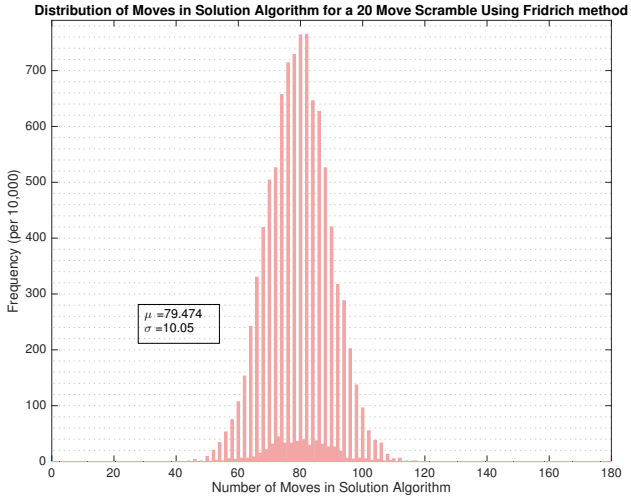

Figure 25: 20-Move Scramble Fridrich Solution Distribution

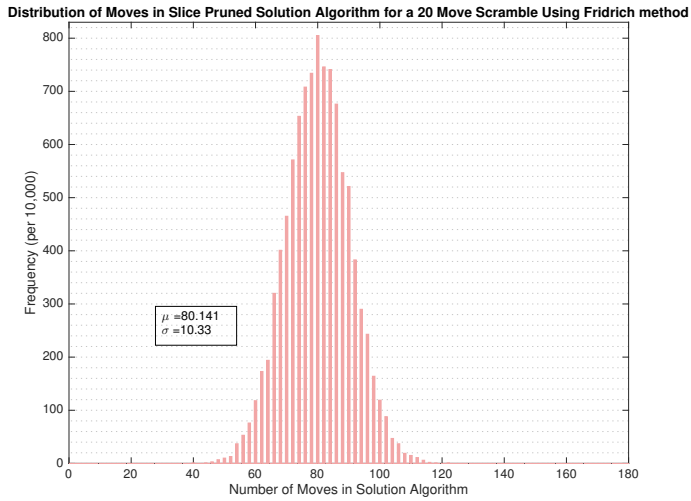

Figure 26: 20-Move Scramble Fridrich 'Slicepruned' Solution Distribution

Much like the CFOP method, the same patterns are illustrated as the number of moves increases, and again it levels off at a similar point:

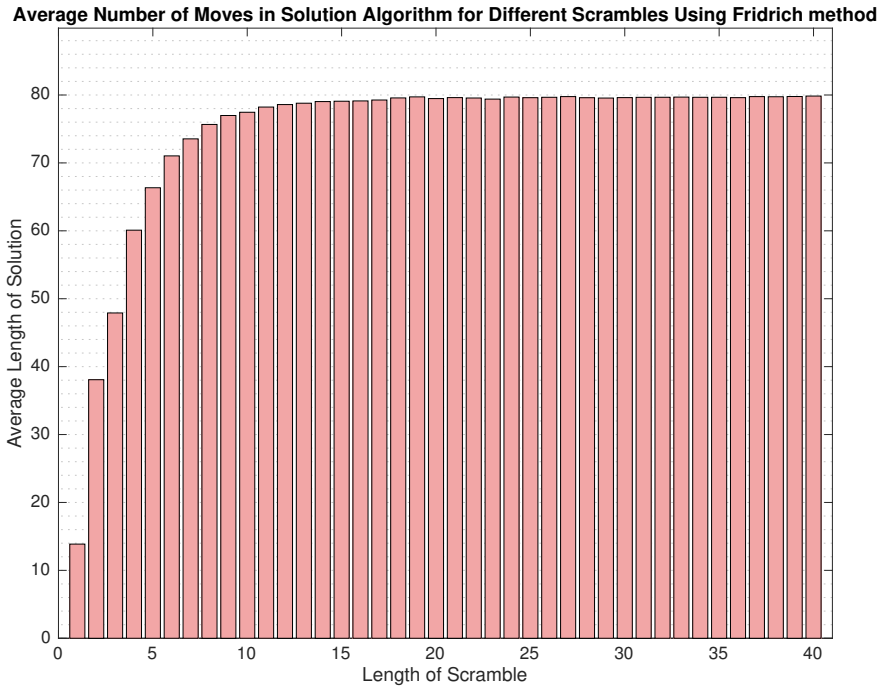

Figure 27: Average Fridrich Solution Lengths

Figures 28 and 29 exhibit a similar trend of the mean moving towards the right, and the standard deviation decreasing, as the length of the scramble is increased:

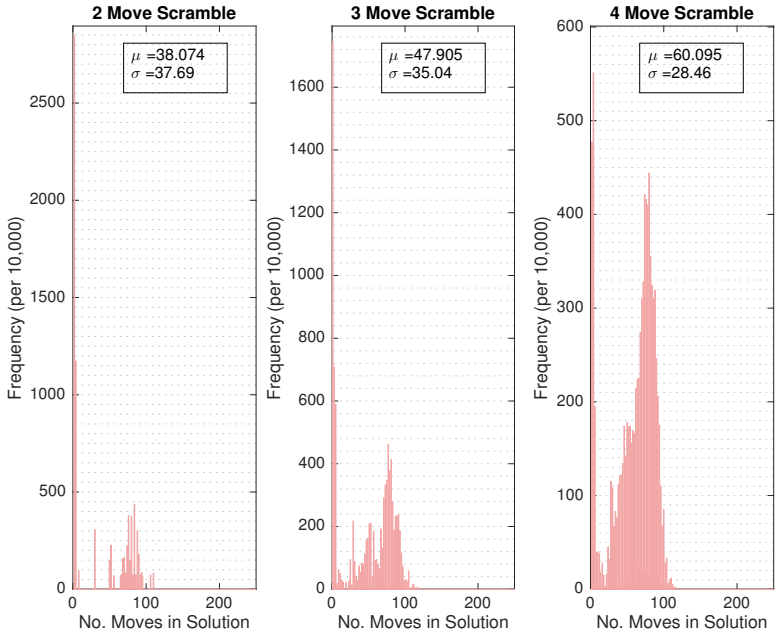

Figure 28: Short-Scrambles Fridrich Solution Distributions

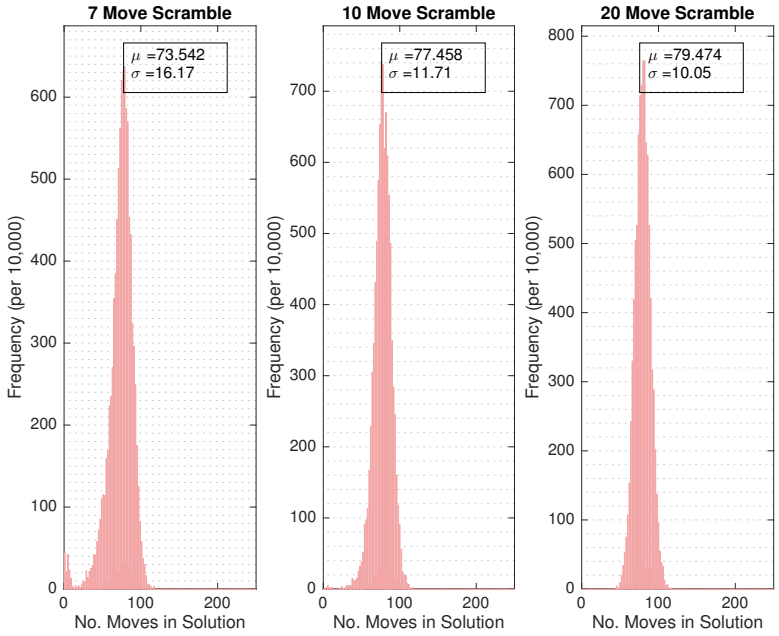

Figure 29: Long-Scrambles Fridrich Solution Distributions

This early levelling off can, again, be attributed to the structure of the search-based method. Unlike CFOP, Fridrich uses only 4 steps, and an analysis of the length of each sub-algorithm is shown here:

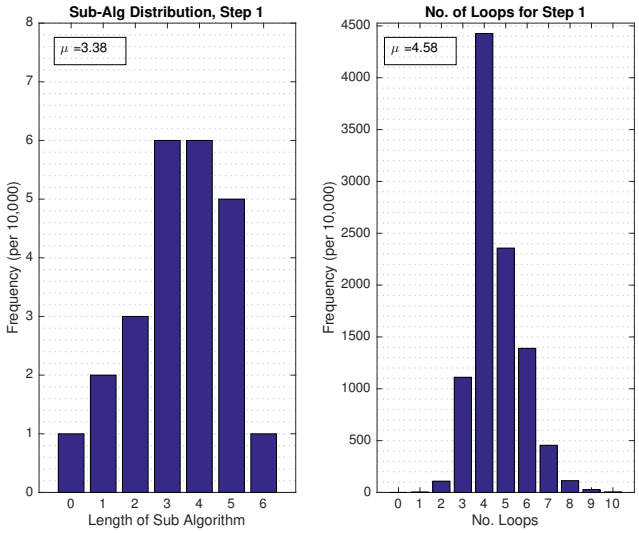

Figure 30: Fridrich Step 1 Move Selection Distribution

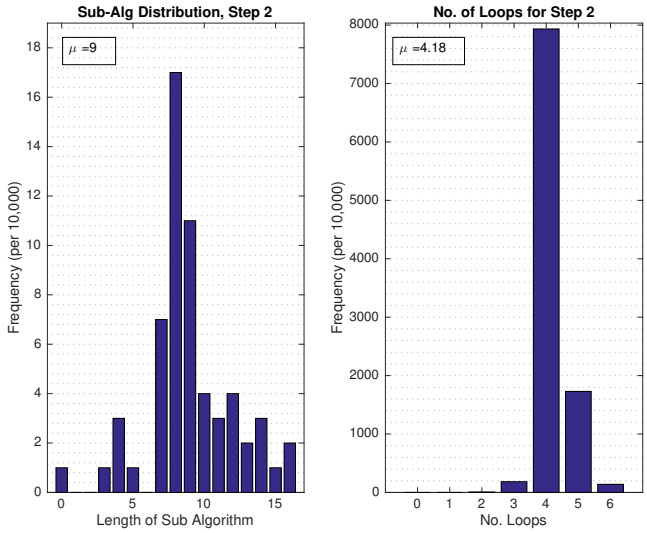

Figure 31: Fridrich Step 2 Move Selection Distribution

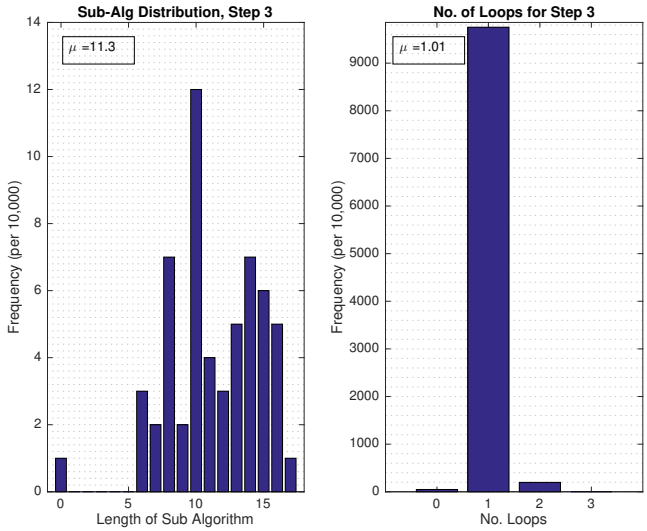

Figure 32: Fridrich Step 3 Move Selection Distribution

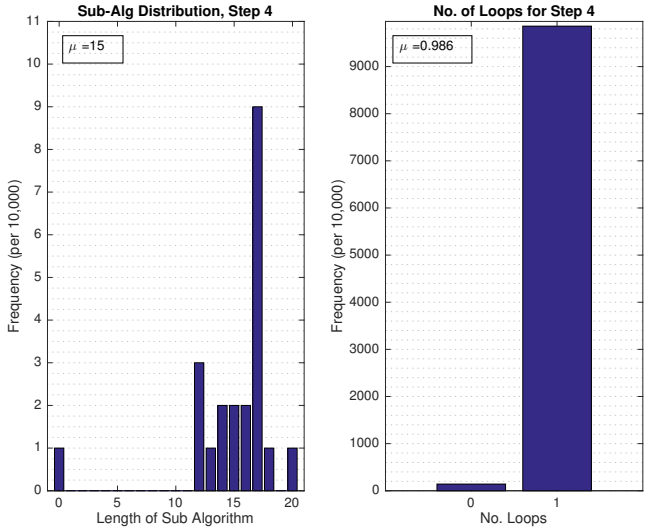

Figure 33: Fridrich Step 4 Move Selection Distribution

This gives us a range of minimum and maximum lengths: 29 and 204, and by taking averages we see that the average length of an arbitrarily chosen solution algorithm comes to 79.3. The discrepancy between this and the actual average is due to the aforementioned differing probabilities of different scenarios.

Finally, figure 34 is an overview of how the length of each sub-algorithm changes as the scramble increases, and where the majority of the moves for each solve come from:

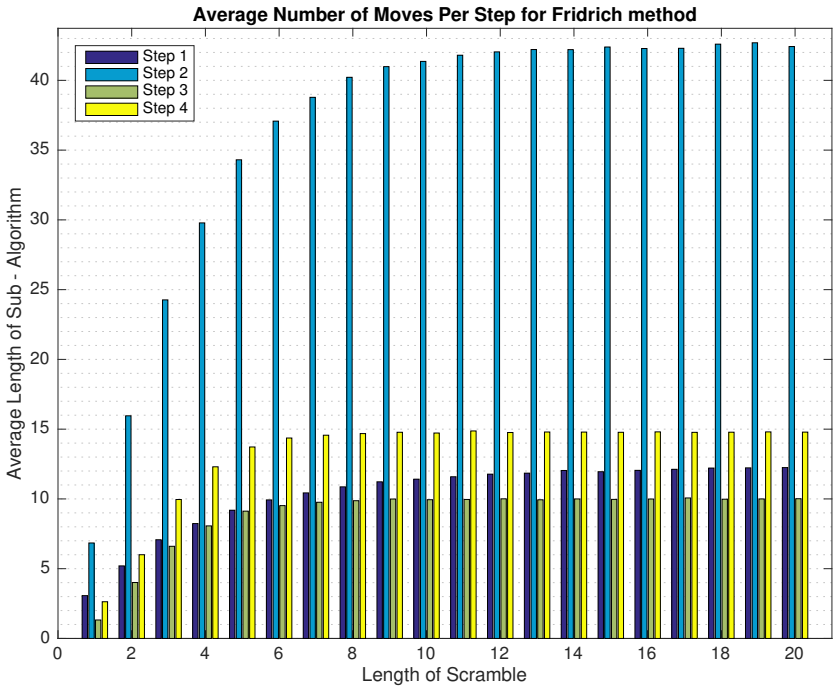

Figure 34: Per-Step Averages for Fridrich

4.1.3 Ortega

I did not see the artefacts that I saw with the CFOP and Fridrich methods when I began to plot results for the Ortega method. This was because the Ortega method makes heavy use of the central slice moves. I could have replaced these with the appropriate outer face turns, but felt that this would unfairly penalise the approach, as it is fundamentally built around those moves.

Again, we see the pattern of mean moving right and standard deviation decreasing. Though, for Ortega it happens a lot earlier, with few low-move scrambles being solved with low-move solutions. This is primarily due to the fact that the Ortega method focusses on placing corners, and utilises lots of central slice moves in order to move pieces around. However, the scrambling algorithms do not make use of

the central slice moves, and so the likelihood of a scramble being luckily undone, as happens frequently with the other two approaches, is more unlikely.

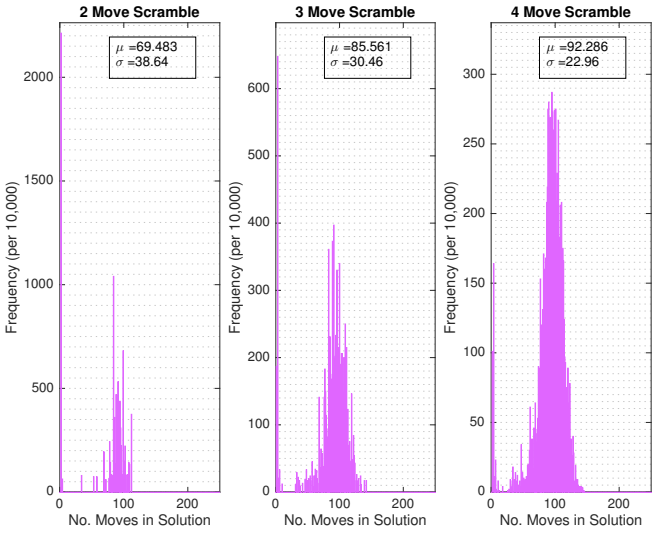

Figure 35: Short-Scrambles Ortega Solution Distributions

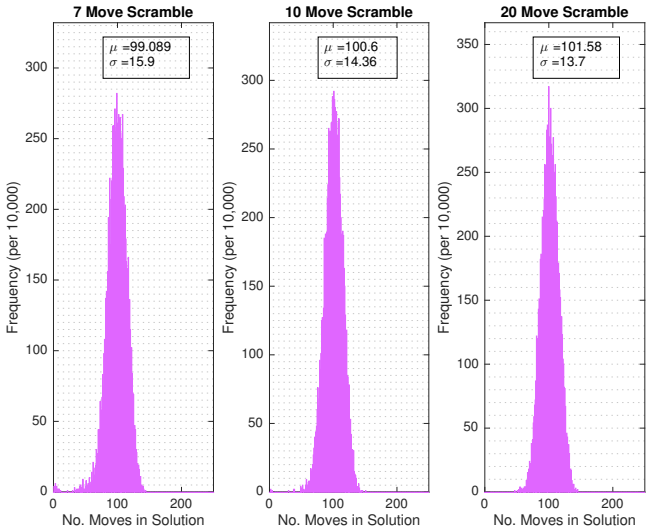

Figure 36: Long-Scrambles Ortega Solution Distributions

Similar to CFOP, the standard deviation of the Ortega method is somewhat

higher than the Fridrich method as there are 6 different steps, with a wide range of choice at each stage, and therefore a higher chance of divergence.

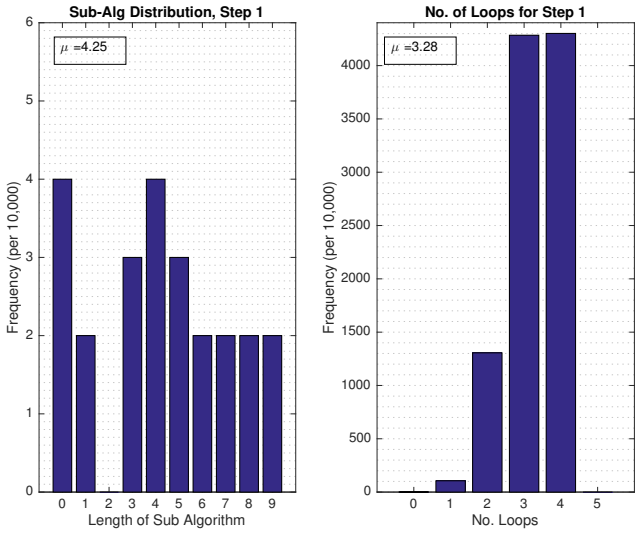

Figure 37: Ortega Step 1 Move Selection Distribution

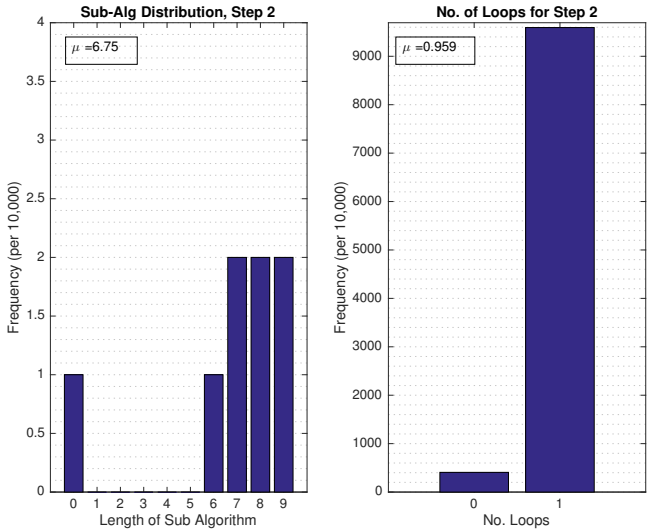

Figure 38: Ortega Step 2 Move Selection Distribution

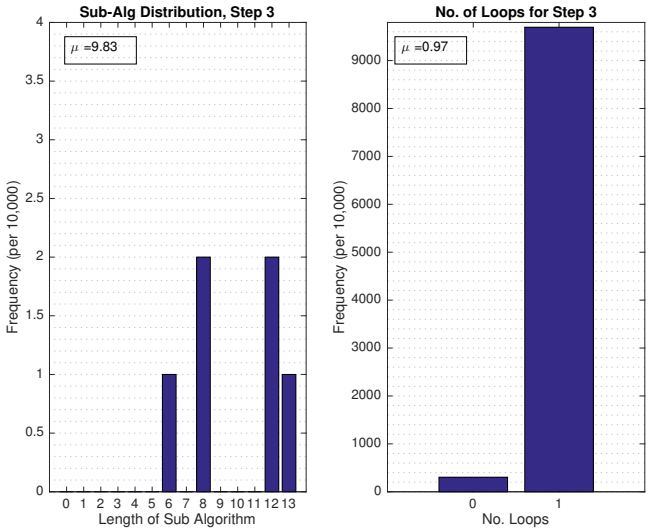

Figure 39: Ortega Step 3 Move Selection Distribution

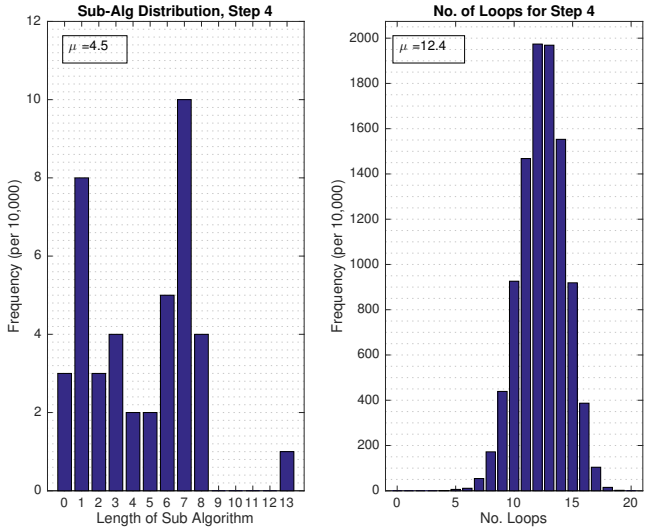

Figure 40: Ortega Step 4 Move Selection Distribution

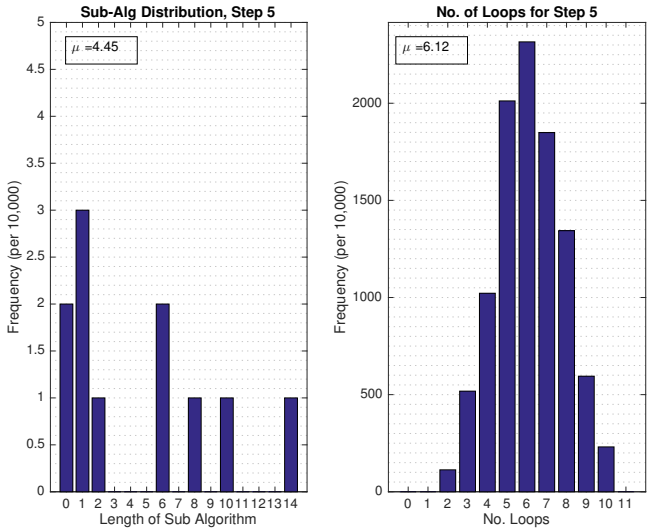

Figure 41: Ortega Step 5 Move Selection Distribution

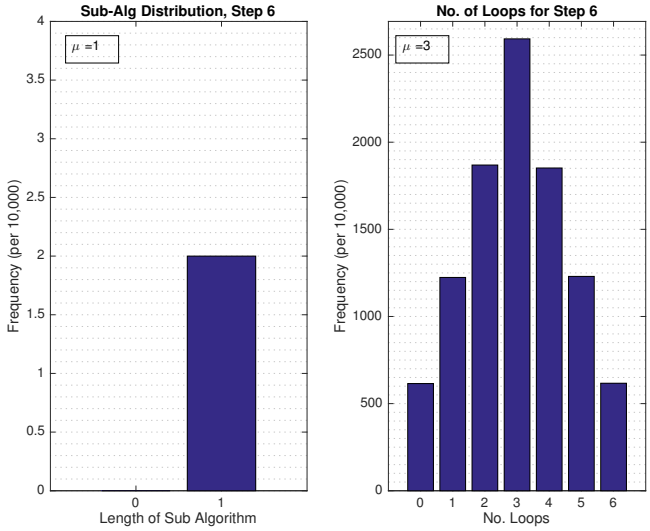

Figure 42: Ortega Step 6 Move Selection Distribution

We can calculate the maximum (464) and minimum (22) solve lengths, as well as the average (116). In general, we see a similar pattern with the number of moves required to solve, and we average to around 101 moves per solve once we get away from the low scrambles.

4.1.4 Comparison

Through the above evaluation, I was able to show a significant decrease in the number of moves required to solve an arbitrarily scrambled cube when using the Fridrich method in preference to the CFOP method. In fact, even at semi-scrambles the Fridrich algorithm produced solutions with fewer moves:

	Average Number of Moves in Solution	
Moves in Scramble	CFOP	Fridrich
1	27.9	13.9
2	68.0	38.1
3	85.0	47.9
4	107.8	60.1
5	117.2	66.3
6	125.1	71.0
7	129.4	73.5
8	132.3	75.7
9	133.7	77.0
10	135.1	77.5
11	135.7	78.2
12	136.3	78.6
13	136.9	78.8
14	137.2	79.0
15	137.8	79.1
16	137.8	79.1
17	138.1	79.3
18	137.9	79.6
19	138.2	79.7
20	138.1	79.5

[3]

In order to show the statistical significance of these results I conducted an independent samples t-test at each length of scramble. For most t-tests the data being

[3]Statistics shown in this table and the next were generated from 1,000,000 randomised scrambles of each length, on an initially solved cube.

tested must be approximately normally-distributed and have homogeneity of standard deviation. The second criterion is not true in our case and so I had to use a slightly different t-statistic which took into account the difference in the variances. The chosen statistic is asymptotically t-distributed and so gives us a conservative estimate for a p-value, though, since we have in the order of 10^6 data points and the same order of degrees of freedom this should still show the significance of my results. I attempted to perform Chi-Squared tests to establish at what lengths of scrambles the solution lengths began to follow a normal distribution. However, despite repeated attempts to achieve this I eventually had to use inspection. The reasons for this can be found in Appendix B. My investigation showed that after 8 moves the results for both CFOP and Fridrich both followed a normal distribution approximately.

Here is the equation I used to calculate the t-values for each scramble length:

$$\mu_C = \text{mean for CFOP algorithm}$$

$$\mu_F = \text{mean for Fridrich algorithm}$$

$$C = \text{length of a solution using CFOP algorithm}$$

$$F = \text{length of a solution using Fridrich algorithm}$$

$$N = \text{number of solutions in a group}$$

$$t = \frac{\mu_C - \mu_F}{\sqrt{\left[\frac{\left(\sum C^2 - \frac{(\sum C)^2}{N_C}\right)}{N_C(N_C-1)}\right] + \left[\frac{\left(\sum F^2 - \frac{(\sum F)^2}{N_F}\right)}{N_F(N_F-1)}\right]}}$$

With the null hypothesis that Fridrich algorithm solves the cube in fewer moves with each solve, and the fact that we are running over such a large number of instances (1,000,000) for each method, a t value of greater that 3.291 would indicate significance at an α level of 0.0005. As you can see from the following table – where t-values are given to 3 significant figures – every length of scramble resulted in a t value greater than 3.291, and this confirms my hypothesis, with a 99.95% confidence level, that the Fridrich algorithm solves an arbitrarily scrambled rubik's cube using significantly fewer moves than the CFOP method.

Moves in Scramble	t-value
8	2020
9	2340
10	2430
11	2500
12	2530
13	2570
14	2590
15	2600
16	2610
17	2620
18	2630
19	2630
20	2630

I ran significance tests comparing Ortega to CFOP and Fridrich, again with an α level of 0.0005, and found that it had a significant decrease in the length of solutions compared with CFOP and a significant increase in the length of solutions compared with Fridrich. For these results please see the appendices.

	Average Solution Length		
Scramble Length	CFOP	Fridrich	Ortega
1	27.9	13.9	29.8
2	68.0	38.1	69.5
3	85.0	47.9	85.6
4	107.8	60.1	92.3
5	117.2	66.3	95.6
6	125.1	71.0	98.0
7	129.4	73.5	99.1
8	132.3	75.7	99.7
9	133.7	77.0	100.3
10	135.1	77.5	100.6
11	135.7	78.2	100.9
12	136.3	78.6	100.8
13	136.9	78.8	101.3
14	137.2	79.0	101.4
15	137.8	79.1	101.5
16	137.8	79.1	101.5
17	138.1	79.3	101.4
18	137.9	79.6	101.7
19	138.2	79.7	101.6
20	138.1	79.5	101.6

This was somewhat unexpected as various resources seemed to imply that Ortega was capable of achieving solution algorithms averaging in the low fifties[4]. I think the reason I was unable to achieve these low numbers is because a lot of these savings are often generated by spotting complex patterns, involving several different pieces, at different stages of the solve. The fact that these patterns are made up of multiple pieces, on different faces, mean there are many different options and variations, and this makes it very hard to efficiently program an approach that can imitate the intuition that humans use when applying this method. Instead a more 'rote' approach must be applied, and we can see that the stages of the Ortega approach where these patterns are spotted (4,5,6) account for a lot of the final length of the solution algorithms.

[4]http://rubikscube.info/ortega.php

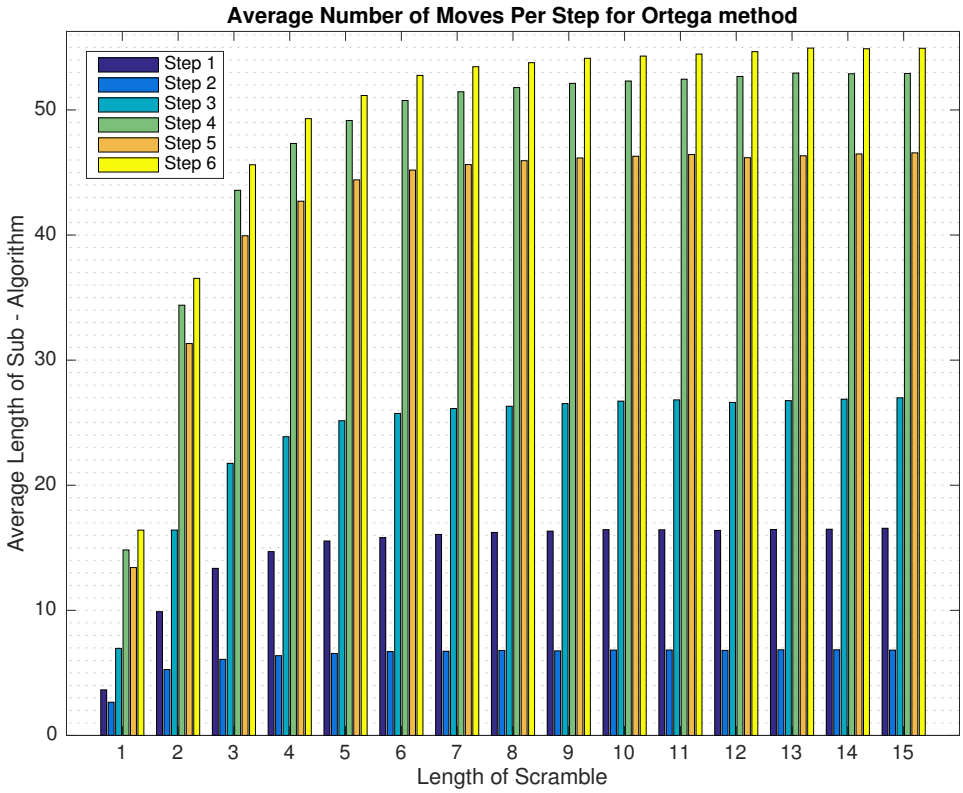

Figure 43: Per-Step Averages for Ortega

Interestingly, the standard deviation of the length of the solves for the Fridrich method was also markedly decreased when compared with CFOP or Ortega.

Both of these phenomena can be observed visually, by overlaying the results of each set of tests for a 20-move scramble on each other. Here we see the distribution taller, and further, to the left for the Fridrich method, compared to CFOP and Ortega.

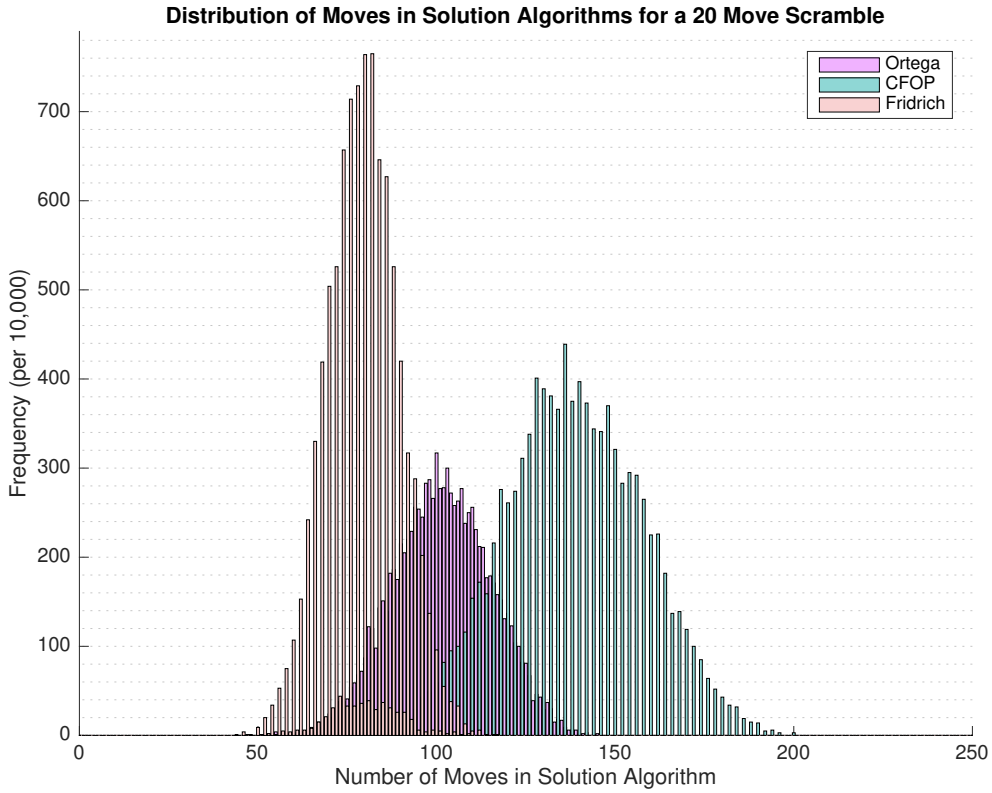

Figure 44: 20-Move Scramble Solution Length Comparison

This can be explained by considering the structure of each solution approach. Whilst CFOP uses 7 different steps and Ortega 6, Fridrich uses only 4, so this limits the scope of the solution algorithms to diverge too much. Furthermore, the distribution of the number of moves in each section is much tighter for Fridrich than it is for CFOP, and closer to Ortega, and so again this means there is less variation in the resulting algorithms with Fridrich.

4.2 Machine Learning Approach

4.2.1 Performance

During the finetuning of the network, I initially evaluated the performance of each of the different network makeups by using the statistics from Weka, including total number of successfully classified instances. Whilst this was a good indicator of

the performance of the model on the training set, it was not necessarily a reliable illustration of how good each model would be at solving a semi-scrambled cube. For example, if the model solved 25% of the total instances of the test data, one might assume that means it would solve 25% of all 10-move scrambled cubes.[5] However, there are various possibilities of the performance of this network: it could be very good at predicting the first 2 to 3 moves to undo a 10-scramble, but incapable of predicting the rest. In this case, the model is useless. Alternatively it might be very good at predicting the last few moves. This would be of more use, as it means the cube could solve 2-move and 3-move scrambles. It could also be a combination of these factors. In order to investigate this, I wrote another analytics class, which would allow me to predict the next move for a scrambled cube, perform the move, and then predict the next move and so on, until the cube was solved or a threshold number of moves was reached. By doing this, I illustrated the performance of the models in solving semi-scrambled cubes.

[5] 10 is the number of moves used in the initial training set.

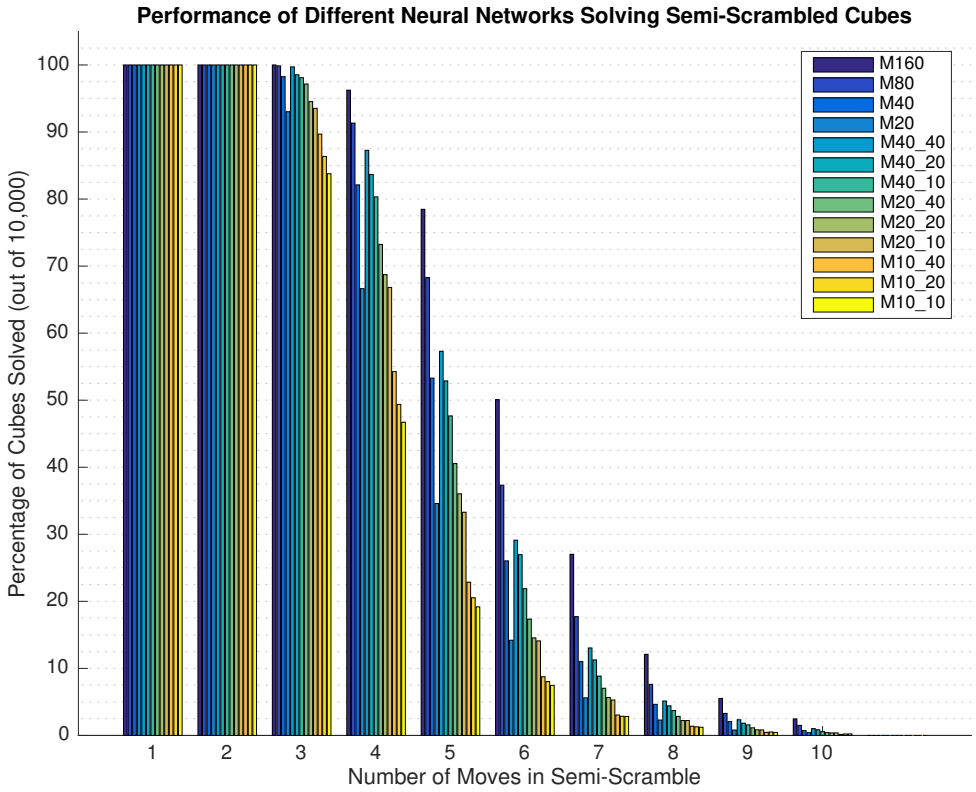

Figure 45: Neural Network Performance [6]

The results showed that the models which had performed best in terms of correctly classifying instances, were still performing best when it came to actually solving a cube.

All of their performances drastically decrease when more than 5 moves away from solving, though the one layer 160-node model still performs well. This decrease in performance is linked to the concept of the disorder of the cube increasing at every stage and, the more disordered the cube the less susceptible it is to being successfully solved by a neural network.

At this point, I considered changing the approach and training a full network for each move away from solved. I think this would have worked extremely well, but decided against pursuing this approach for 2 reasons: firstly due to the time

[6]The statistics shown in this graph were generated by running 10,000 random scrambles for each length of semi-scramble.

constraints of the project; but more crucially, it would reduce the flexibility and extensibility of the approach. By maintaining a training set with all the scramble sizes in together, it meant that a cube which had been scrambled with a 3-move or a 5-move scramble could be passed into the same network. As one of my extensions involved trying to get a cube into a position where it was almost solved, I preferred to maintain a flexible solution in which I did not have to state how many more moves would be required, as this would not be feasible the majority of the time.

Instead, I attempted improve the performance of the network with alternative methods. I observed the activity of the fine-tuned network when attempting to solve 5-move scrambles on my graphical representation. I noticed that when it failed, it would do so in a way that meant it ended in a limit-cycle of two states, repeating one move followed by another repeatedly until the maximum number of moves was reached. I wrote a function as part of the neural network solving class, which would detect if a cube had entered a limit-cycle, or toggle, state. Then, in order to avoid this I had the network select the second most likely move. This yielded no significant improvement.

This was not surprising as for the cube to end up in a toggle situation it must have taken an incorrect turn earlier on. I experimented with which move was most likely to cause a toggle by backtracking the moves made to each point and then selecting the second most likely move and continuing the solve from there. I still kept track of the number of moves being made, but set the cap higher in order to give the cube a chance to backtrack and solve itself. First, I looked at cases where it would backtrack to the start; and secondly, when it would backtrack to the move which caused it to toggle. From the results it seems that the problem was more frequently the first move, as this scenario, where the solver backtracked to the first move, resulted in more cubes being solved.

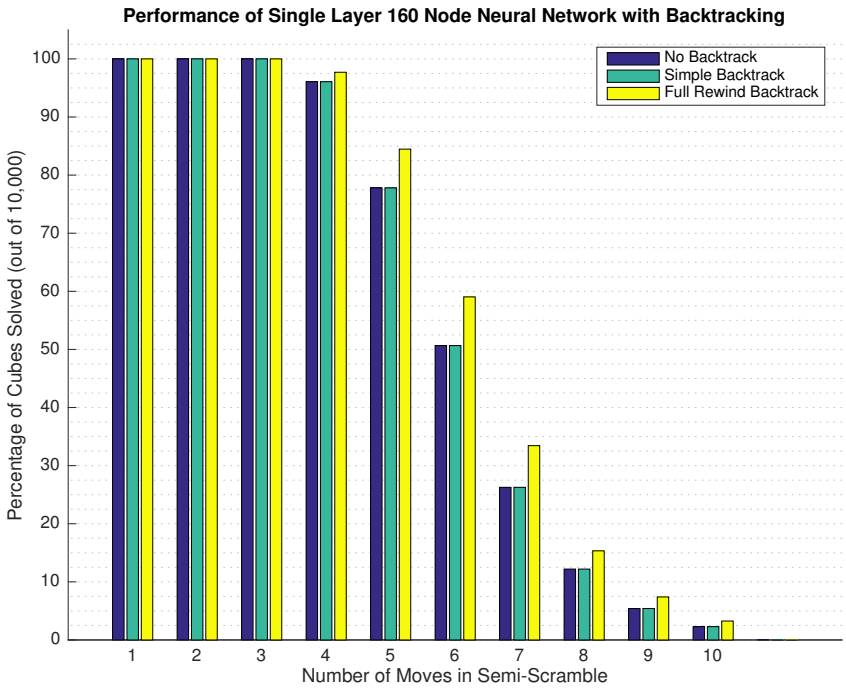

Figure 46: Backtracking Performance

I decided to implement a second backtrack because the second option of backtracking to where the toggling began was still causing an improvement occasionally. If I allowed the network to attempt both then I should be able to see a greater improvement. This hypothesis seemed reasonable, if the problem was not the first move, and I backtracked to the first move and performed the second most likely move, the cube would often return to the original state and continue the solve as it did the first time anyway, ending in the same limit-cycle. I therefore implemented the *neuralsolve* method to first attempt the solve, if there was a toggle, then return to the beginning of the solve and try the second most likely move. If it toggled again, then would backtrack to the move before the toggle and attempt the second most likely move here instead.

This did not yield as significant an improvement as I had hoped; in fact, occasionally it would perform worse than the full rewind approach. I thought this result was probably due to random scrambles being used, which were generated each time. To check this, I tested both approaches on the same 10,000 randomly generated scrambles.

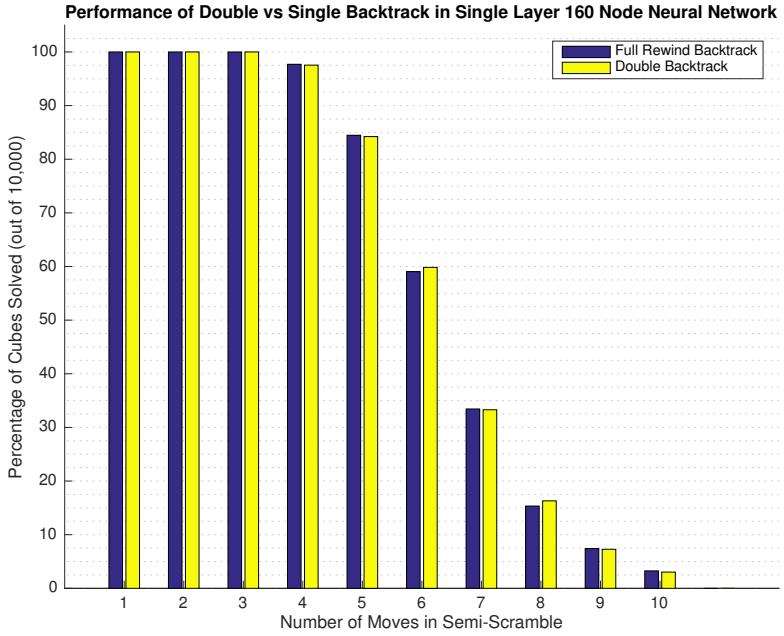

Figure 47: Further Backtracking Performance

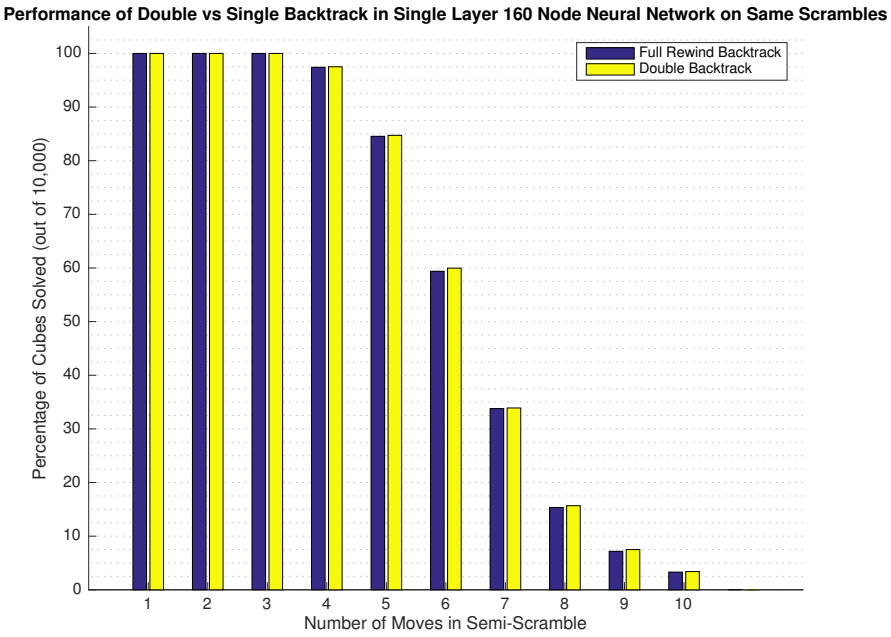

Figure 48: Augmented Backtracking Performance

As we can see there is a slight improvement at every step after step 3. I decided to leave this as the final structure of the neural solve approach.

4.2.2 Controlled Experiment

One of my aims was to see if I could build a machine learning approach which could solve a semi-scrambled Rubik's cube better than a human could perform this task. In order to measure this, I first designed an experiment to test how well humans performed at the task. I recruited 20 volunteers. I presented each volunteer with a cube, which had been turned a certain number of times, from 1 move up to 10 moves. I initially considered randomising the order in which the participants were given the cube, but instead allowed them to do it incrementally, presenting them with a 1-move, then a 2-move, and so on. I did experiment with 2 separate volunteers outside of the controlled experiment who both agreed that they found it significantly harder to solve a 4-move scramble, having not first seen a 3-move scramble. Whilst I appreciate this is not a particularly rigorous way of testing this as a hypothesis, I wanted the humans to perform to the best of their ability and, equally, I was limited both in time and participants, so I decided this was an appropriate sacrifice to make.

Method

Each participant was given 3 separate cubes, which had been scrambled with randomly generated 1-move scrambles, then 2-move scrambles, all the way up to 10-move scrambles. These scrambles were randomly generated. Every scramble was recorded. The participant was given the same number of moves to solve the cube as was used to scramble, and as long as they desired to think about it. If they succeeded in solving the cube in the allocated number of moves, this was recorded as a success, otherwise it was a failure. The age, gender, ethnicity and occupation of the participants was recorded, as well as their familiarity with Rubik's cubes, in order to try and control for external forces on the results.

Results

The results of all the participants were triaged together to obtain the percentage of cubes solved from X-moves away. These results are plotted below.

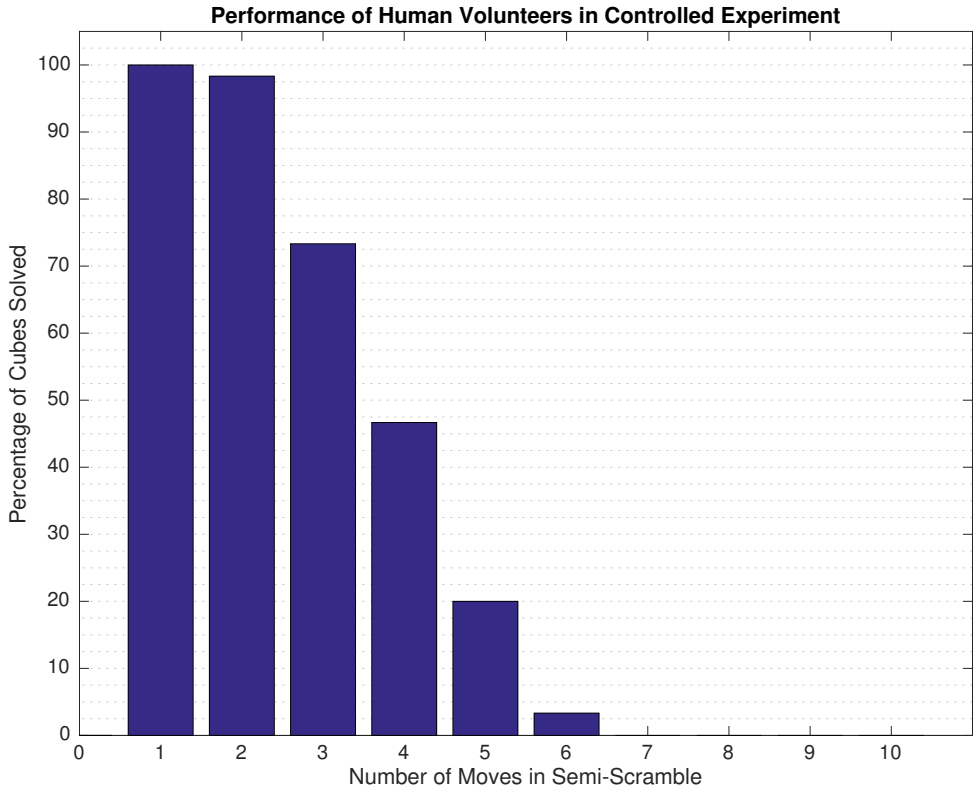

Figure 49: Human Performance

The performance of human participants drops off rapidly as the number of moves increases, with only 2 people being able to solve anything further than 5 moves.

I attempted to do discriminant analysis testing using SPSS to show that the human participants performance was not affected by the factors I had recorded. Unfortunately, the relatively low number of participants meant that no significant conclusions could be drawn. Since the experiment was attempting to show that the performance of humans differed from the performance of the neural network, variation within the results of the participants was not critical.

4.2.3 Human vs Computer

When comparing the performance of humans against the neural networks, one could simply look at the percentage of X-move scrambled cubes solved by humans and compare this to the figures gleaned from evaluation of the neural networks. From

those results (see previous figures), we can see that the drop off in performance as the scramble length increases is less severe for the neural network than it for human participants, and the neural network outperforms the human volunteers at every stage. However, there is a possibility that the scrambles given to human participants were more challenging, and this is why they were performing worse than the neural network. To show this was not the case, I tested the neural network using identical scrambles to the human tests and plotted the results.

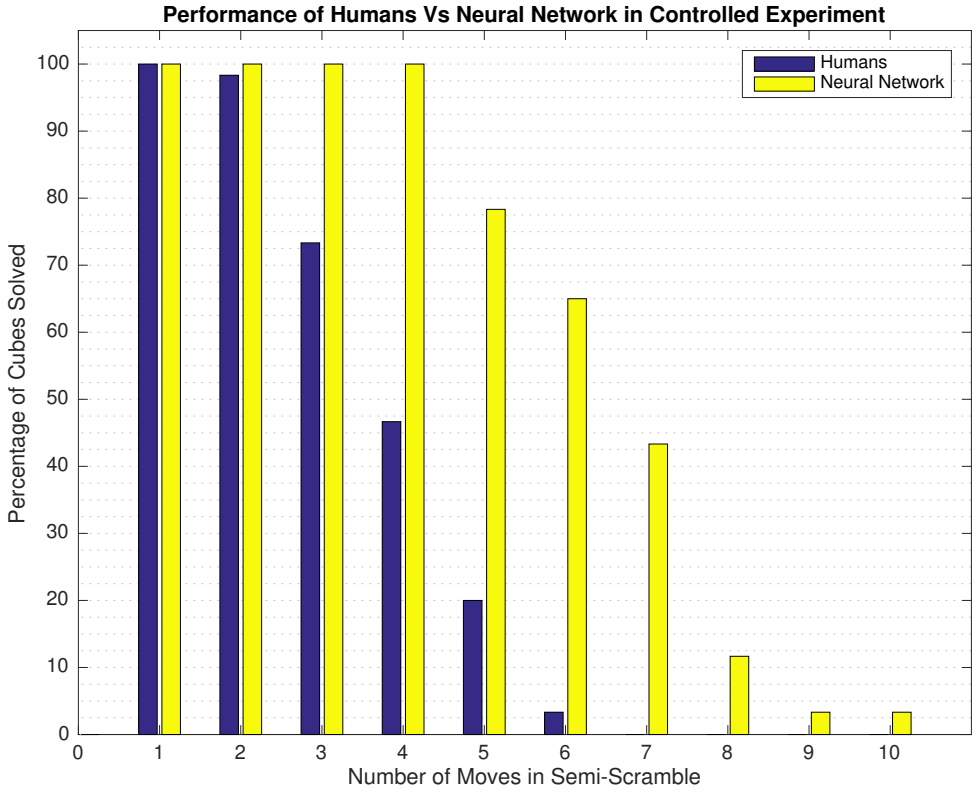

Figure 50: Human vs Computer Performace

Even tested on the same scrambles, the neural network outperforms the participants at every level. Moreover, I compared to see how many of the scrambles that the humans could not solve, the neural network could solve and vice versa, and found that there were only 2 cases where the human solved a cube the neural network didn't, compared to 160 conversely. For completeness I carried out t-testing on the results of the experiment and found that the neural network performed significantly

better than human participants at this task. **This confirmed the hypothesis of the experiment, and a large goal of my project, that a neural network can be trained to outperform humans (with limited exposure to Rubik's cubes) at solving semi-scrambled Rubik's cubes.** These results should be taken with some apprehension as it is difficult to draw conclusions about humans' general performance at a task from a very small subset, and, moreover, a subset who had not all spent significant time with Rubik's cubes. Furthermore, those that had spent time with Rubik's cubes had probably not spent significant time attempting to solve semi-scrambled cubes by undoing the scrambles. These facts, coupled with the fact that humans' competing in Rubik's cube solving competitions frequently record impressively low solutions[7] even for fully-scrambled cubes, implies that given time to train, the performance of humans' could greatly increase compared with my sample group.

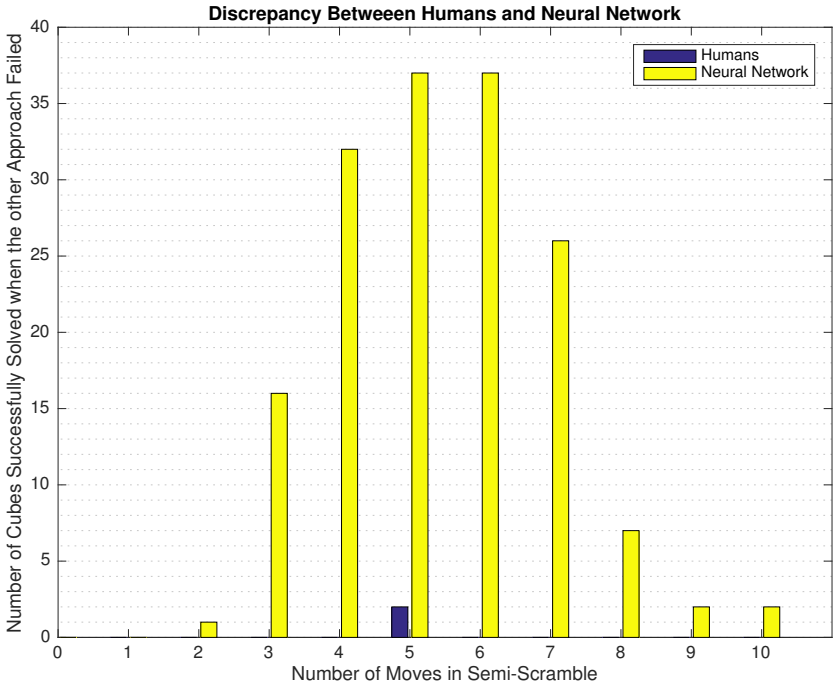

Figure 51: Human vs Computer Discrepancy

[7]Recent competitions: `https://www.worldcubeassociation.org/competitions`

4.3 Evaluation Against Success Criteria

I shall address each of my success criteria here:

1. *Write a Rubik's cube simulator*

   This was successfully achieved early in the project, using Java to build an object-oriented class-based approach and keep associated methods paired with the cube.

2. *Write software to solve a scrambled cube using a search-based method*

   This was achieved.

   - Implement traditional human-friendly algorithms (CFOP/Fridrich). This section was achieved and fine-tuned to return the most parsimonious algorithm from each method.
   - Implement more advanced human-friendly algorithms (Roux/Petrus/ZZ) This section was partially achieved. As stated in my preparation, it became apparent that implementing these specific algorithms would not be valuable. However, I did implement the more complex Ortega approach successfully.
   - *Extension:* Implement advanced, more computer-focussed, search-based algorithms (Thistlethwaite/Kociemba). This section was not completed.

3. *Evaluate the performance of three different methods from (2).*

   This section was achieved with thorough evaluation of all solution methods implemented, including custom test-scripts written in Java and numerous plotting functions written in MATLAB.

4. *Use mathematical analysis of permutations of Rubik's cubes, to show when it becomes infeasible to store every possible permutation of the cube.*

   This section was achieved by illustrating the storage requirements necessary, both in an optimal approach and in the chosen approach.

5. *Use a machine learning approach to train a neural network to solve a semi-scrambled cube.*

This section was achieved using Weka to train various different networks, and then selecting one to fine-tune.

6. *Evaluate the performance of (5) as the number length of semi-scramble is increased.*

   This section was achieved by writing testing scripts and MatLAB plots.

7. *Evaluate the performance of (2) and (5) on semi-scrambles.*

   This section was achieved using the aforementioned scripts, and combining the resulting statistics and plots.

8. *Extension:* Introduce a machine learning approach to the initial part of the solve.

   This section was not achieved due to time constraints, but the approach I would have taken is outlined in the conclusion.

9. *Extension:* Develop a hybrid approach using (8) and (5) to solve a fully scrambled cube.

   This section was not achieved as (8) was not achieved.

   - Contingency: develop hybrid approach using (2) and (5)

     This was attempted, but no meaningful results were obtained and so it was omitted.

10. *Extension:* Evaluate the performance of hybrid solution against (2)

    This was not completed.

11. *Extension:* Build graphical representation of (1)
    This section was achieved and rationale for not further developing was explained.

    - Contingency: Use open-source simulator.
      This section was deemed unnecessary, due to the previous success.

Chapter 5

Conclusion

5.1 Successes

The project, as a whole, was a success. I achieved almost all of my core objectives and successfully implemented some of my extensions. I was particularly pleased with the solutions I devised for the complex problem of indexing different states, such as building the unique integers by observing outwards facing cells for the OLL step. I was able to reuse this idea to tackle similar problems throughout the project. I felt the addition of the controlled experiment to the project gave it another level of interest in demonstrating how neural networks can be trained to perform tasks which humans find very difficult at a higher level of success. It also gave me tangible results to compare those from the neural network to. Finally, I was pleased to implement a graphical representation of the cube, as it made my project more accessible when explaining it to others and provided a useful visual tool for witnessing how the solution approaches worked and how they differed from each other. It also made it possible to illustrate how the neural network was approaching a solution.

5.2 Failures

Unforeseen difficulties were outlined in my progress report and earlier in this document. The main failing of the project was not committing sufficient time to the machine learning approach. This was mainly fuelled by a greater interest in the search-based approach and consequently this was very comprehensive. Due to my

lack of prior experience with machine learning I should have allocated more time to investigate different ways of selecting features as well as properly researching how to use Weka's built-in serialisation and avoid the problem of not being able to serialise models built using the explorer. At the very least I should have allocated more time to train a wider range of different models. This lack of variety of models and ability to further train existing models meant that I was hampered in what I could achieve, and had to look at creating workarounds instead of improving the models' pure performance. In a way, this helped me to understand ways of increasing models' performance without further extensive training, such as the backtracking extension; however, the project may have been more fulfilling had I investigated more extensive training methods as well as trained networks further. The only other failure of note, which eventually worked to my advantage, was the initial peculiarity in the data when evaluating the performance of CFOP and Fridrich. I list this as a failure as it transpired to be human error on my part, but in reality this allowed me to create extensive analysis programs and display exactly what was going on at each point of the solve, which I potentially would not have done without this mistake.

5.3 Alternative Approaches

Were I to start this project from scratch now, the biggest change I would make would be to draw up a detailed training plan for the neural network much earlier than I did, and begin to carry it out as soon as I could in order to give myself the most possible time to fine-tune and select appropriate models. Additionally, I would have carried out further research into different approaches to training models to see if other frameworks offered more varied models. In a similar vein, I would have sought to obtain more computing power to run these tests more efficiently. Additionally, with hindsight I may have approached the controlled experiment slightly differently and attempted to survey a wider range of people. In ideal circumstances I would have looked to attempt to test people with significant experience in solving Rubik's cubes in few moves. Realistically it would be very difficult to carry out this kind of experiment as part of a Part II Project.

5.4 Continuing the Project

I would implement more search-based algorithms, and obtain a proper picture of how more complex block-building approaches perform. I would also further refine the algorithms I did implement; in particular, to see if there were further ways of significantly cutting down moves in the early, more intuition-based, stages of the solves. As well as the aforementioned changes to the machine learning approach, I would attempt to build a model to target the initial stages of the solve using a reinforcement learning approach and rewarding the model for the bigger size blocks it achieved. I would then look at trying to join this with the existing model I built, to have an entirely machine learning based approach to solving an arbitrarily scrambled cube.

Bibliography

[1] Erik Demaine, Martin Demaine, Sarah Eisenstat, Anna Lubiw, and Andrew Winslow. The mathematics of the rubik's cube. http://web.mit.edu/sp.268/www/rubik.pdf, 2009.

[2] Daniel Duberg and Jakob Tideström. Comparison of rubik's cube solving methods made for humans. http://www.diva-portal.org/smash/get/diva2:812006/FULLTEXT01.pdf, 2015.

[3] Katyanna Quach. Wanna get started with practical ai? check out this chap's rubik's cube solving neural-net code. https://www.theregister.co.uk/2017/09/25/rubiks_cube_neural_network, 2017.

[4] Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. God's number is 20. http://www.cube20.org, 2010.

[5] Daniel Ross. *Rubik's Cube Best Algorithms*. Independent, 2017.

Appendix A

Project Proposal

This first appendix contains the original project proposal.

Thomas Davidson
Corpus Christi
tjd45

Part II Computer Science Project Proposal

Comparing a Search Based Method to a Machine Learning Approach to Solving a Rubik's Cube

12/10/2017

Project Originator: *Thomas Davidson*

Resources Required: See Resources Required Section overleaf

Project Supervisor: *Sean Holden (sbh11)*

Signature:

Director of Studies: *David Greaves (djg11)*

Signature:

Overseers: *Richard Mortier (rmm1002)* and *Andrew Rice (acr31)*

Signatures:

Introduction and Description of the Work

This project involves the comparison of a search-based computer program for solving a Rubik's cube to a machine learning approach. The search-based method will be implemented to use traditional human methods of solving the cube such as CFOP/Fridrich (a traditional beginners' solving technique) and the ZZ method (a more recent approach which can use fewer moves), as well as other, less human-friendly, approaches too, such as Thistlethwaite and Kociemba which are both based on group theory and explained nicely here[1]. For more information about these solution techniques as well as others which will be implemented please see the website below [2] which gives a useful overview of differing solution approaches. In general, most of these approaches use a wide variety of different algorithms to transform the cube from one state to another, eventually finishing with a solved cube. The machine learning approach will attempt to use a large training set of random scrambles (scramble: a set of turns performed on a solved cube) to see if a program (likely a neural net) can be trained to solve a cube from X number of moves away.

Then the performance of both approaches will be tested by providing each of the programs with a cube which is X number of moves from being solved. The performance of differing solving methods will also be evaluated. Performance will be based on whether the cube is solved, time to generate the solution, and the number of moves in the solution. From here I will attempt to implement a machine learning approach for the beginning of the solve and see if it is possible to build a program which can solve the cube end to end without having to hardcode any algorithms. Failing this, I would try to build a search based method which gets the cube into a situation where the machine learning approach can take over. I would then also compare the performance of this solution with a purely search based version.

A similar form of machine learning approach is described in the article below [3] and this is where much of the inspiration comes from. I want to build on this though and attempt to eventually build something that can deal with an arbitrarily scrambled cube rather than only one that had only been scrambled using an X turn scramble. Whilst several variations of search based method solutions already exist [4] I think it would be valuable and challenging to develop my own and would also allow me to perform evaluation of different solving techniques against each other, similar to that performed in this study[5].

Resources Required

Personal laptop for majority of development: Mid 2014 15-inch MacBook Pro with 2.8GHz Intel Core i7 Processor and 16GB of RAM. Will also use an old desktop to run some of the training sets on. If this is not sufficient then I plan on investigating the possibility of using the Wilkes cluster to run some of the training on.

Starting Point

I have had brief prior experience of attempting to code a Rubik's cube simulator in Python (purely non-graphical) and a long standing interest in Rubik's cubes and the mathematics surrounding them, but have never undertaken a serious programming project involving them.

Substance and Structure of the Project

The initial part of the project would involve writing a simulator for a Rubik's cube and establishing rules for representing the state of the cube and operations that can be performed on it, as well as algorithms. Whilst open source simulators do exist I think writing my own simulator would allow me more freedom in designing algorithms later on as well as being an interesting challenge into how best to represent the cube and transformations of it. Initially I would develop this as a non graphical simulator with a text based output to keep it simple. As this should be a fairly straightforward task I will probably develop a version in Java and in Python for ease of use later in the project

Next would involve writing the software to take a scrambled cube as input and output a solution algorithm. The solution algorithm would be generated using a search based method based on traditional solving techniques. I would begin with simple to learn (for humans) algorithms such as CFOP/Fridrich (described above and in [2]) before moving onto more complex approaches (Roux/Petrus/ZZ) and finally looking at implementing very complex search based approaches as an extension (Thistlethwaite/Kociemba [1]) As I have most experience programming in Java I would develop this part of the project using it.

I would then use a machine learning approach to train a neural net to 'solve' a Rubik's cube from a certain number of moves away. Clearly at low numbers this could be achieved quickly and efficiently using a lookup table so I would perform some mathematical analysis to work out at what point using a lookup table becomes infeasible in terms of memory and time requirements and thus show why a machine learning approach is a sensible one to use. For the training I would use repeated random scrambles of set numbers of moves and then present these scrambles, with the corresponding state of the cubes, in reverse as solutions to a semi-scrambled cube. For example if you denote the state of the solved cube as C and perform a two turn scramble: RU', where R represents one 90 degree clockwise turn of the right face and U' represents one 90 degree anti-clockwise turn of the upper face. And say this transforms the cube from C to C*, by inverting the two turn scramble, that is reversing every move as well as the sequence, we provide a solution algorithm from C* to solved cube C of UR'. Features that willl be incorporated to help the training will be the location of edges and corners. By doing this with a low number of moves and increasing it after performance plateaus, it should allow the net to predict the next best move until the cube becomes solved. I would test this hypothesis by testing with unseen scrambles of a set number of moves and monitoring the results. I would hope to be able to achieve near to 10 moves, as this is where human performance begins to tail off. However by performing the training incrementally and gathering results I will be able to analyse the performance of this approach and see after what number of moves it begins to struggle. This section of the project will probably be written in Python due to the high availability of machine learning libraries. After discussion with my supervisor we deemed it acceptable to have this different from the search based method, as well as leaving the potential opportunity to develop it in Java if it becomes apparent this would be more appropriate.

At this point I would evaluate the performance of the machine learning approach with the search based method by presenting both with set number of move scrambles. Would look at the compute time for the solution algorithm, whether the cube is solved and how many moves are involved in the solution algorithm. I would do this from 1 move up to however many the machine learning approach seems to be able to cope competently with.

From here I would begin to look at what are potentially extensions to the project. I would like to try and introduce machine learning to the initial part of the solve, potentially by taking inspiration from the Roux method (this groups together blocks

of cells (individual constituent squares of the Rubik's cube, there are 3x3x3x6=54 on a standard cube)) as this would allow for a reinforcement learning approach as groups of colours would be simpler to implement as an exploration heuristic. The problem with trying to implement a traditional machine learning approach here would be a lack of available training cases as would require a lot of solves inputted into the system to make it possible to train anything meaningful. However by using reinforcement could use a more random approach with a scrambled cube and use the reinforcement learning to make progress. In an ideal world it would be possible to plug a machine learning solution to the beginning of the solve into the earlier mentioned machine learning approach to the end of the solve. As this seems unlikely to be that straightforward, as a contingency plan, I would look at incorporating a search based method to solve a cube up to a level where it could be tackled by the earlier machine learning approach. The performance of this hybrid method with the pure search based method would then be evaluated on fully scrambled cubes.

A final extension that I would very much like to implement would be a graphical representation of the cube simulator. I think this would be a really useful exercise in developing my graphical programming skills, as well as allowing users of the system to better visualise what is going on with the solving. If it happens that this is not feasible I would look at using David Whoggs open source MagicCube [6], and adapting my code to communicate with it.

In summary, the project has the following main sections:

1. Write a Rubik's cube simulator
2. Write software to solve a scrambled Rubik's cube using a search based method
    - Implement traditional human-friendly algorithms (CFOP/Fridrich)
    - Implement more advanced human-friendly algorithms (Roux/Petrus/ZZ)
    - *Extension:* Implement advanced more computer focussed search-based algorithms (Thistlethwaite/Kociemba)
3. Evaluate the performance of the three different aspects of (2)
4. Use mathematical analysis of permutations of a Rubik's cube to show when it becomes infeasible to store every possible permutation of the cube after X moves

5. Use a machine learning approach to train a neural net to solve a semi-scrambled Rubik's cube (semi-scrambled: a set number of moves away from being solved)
6. Evaluate the performance of (5) as the number of moves away from being solved is increased
7. Evaluate the performance of (2) and (5) on semi-scrambles
8. *Extension:* Introduce a machine learning approach to the initial part of the solve
9. *Extension:* Develop a hybrid approach using (7) and (5) to solve a fully scrambled cube
    - Contingency for failure: develop a hybrid approach using (2) and (5)
10. *Extension:* Evaluate the performance of a hybrid solution against (2)
11. *Extension:* Build a graphical representation of (1)
    - Contingency for failure: Use open source simulator[6]

Reference

[1] *Description of Thistlethwaite and Kociemba approaches to solving.* (`https://en.wikipedia.org/wiki/Optimal_solutions_for_Rubik%27s_Cube`)

[2] *Article describing various different solving approaches for a 3x3x3 Rubik's cube.* (`https://www.speedsolving.com/wiki/index.php/3x3x3_speedsolving_methods`)

[3] *Article describing the use of a neural net to solve a rubik's cube.* (`https://theregister.co.uk/2017/09/25/rubiks_cube_neural_network`)

[4] *Example Rubik's Cube Solver which offers basic and close to optimal solve.* (`https://rubiks-cube-solver.com`)

[5] *Comparison of Rubik's Cube Solving Methods Made for Humans*, D. Duberg and J. Tideström, Degree Project, Royal Institute of Technology, Stockholm, Sweden. (`http://www.diva-portal.org/smash/get/diva2:812006/FULLTEXT01.pdf`)

[6] *Open Source Rubik's cube visualiser.*
(`https://github.com/davidwhogg/MagicCube`)

Success Criteria

The following should be achieved:

- Core
  - Simulate a 3x3x3 Rubik's Cube
  - Develop a search-based method which outputs a solution algorithm when given a scrambled cube as input based on:
    - Basic human algorithms
    - Advanced human algorithms
    - Advanced computer algorithms
  - Evaluate the performance of the different types of algorithms to solving scrambled cubes
  - Train a neural net to output solution algorithms to a set number of move 'semi-scrambles'
  - Evaluate the performance of the neural net as the number of moves is increased
  - Evaluate the performance of the search-based method versus the machine learning approach
- Extension
  - Use a machine learning approach to complete the early parts of the solve
  - Build a hybrid machine learning approach to solve a fully scrambled cube
  - Build a graphical representation of the simulation
  - Extend the simulator to deal with NxNxN Rubik's Cubes
  - Extend the search based solution to deal with larger Rubik's Cubes
  - Extend the machine learning approach to deal with larger Rubik's Cubes

Timetable and Milestones

Weeks 1 to 3 : 20th October - 10th November

Initially consideration of different ways of storing the state of the cube and algorithms (I use algorithms here to refer to a series of face turns of the cube) and then development of the Rubik's cube simulator in order that it can display an interactive textual model of a Rubik's cube which changes its state when given turns or algorithms as input. Implementation of the basic human-friendly algorithms, namely CFOP/Fridrich to begin with.

Milestones: Clear documentation of the data structures and format the cube simulator will take. Simple test code which will display the textual representation of the Rubik's cube and how it changes when given turns as input. More robust code which will solve an arbitrarily scrambled Rubik's cube using the CFOP/Fridrich approach and output the solution algorithm (a series of face turns represented textually).

Weeks 4 and 5 : 10th November - 24th November

Implement the more complex human-friendly approaches: Roux/Petrus/ZZ and document how they differ from the previously implemented methods. Evaluate the performance of the simple approaches versus the more complex. Evaluation will take place in the form of time taken to generate solution algorithm and the total number of moves needed to solve. This should be presented in a clear and understandable format, most likely graphical.

Milestones: Robust code to solve the cube using more advanced methods. In-depth analysis of the performance of the various different approaches against one another.

Weeks 6 and 7 : 24th November - 8th December

Complete mathematical analysis to indicate the appeal of a machine learning approach by showing when it becomes infeasible to store all possible permutations of the cube and their corresponding solution algorithms. This period coincides with the end of term and Varsity Ski Trip hence the significantly lighter workload, will also be a chance to tidy up and of the previous sections that have been completed to this point.

Milestones: Detailed analysis of some of the mathematics behind brute force approaches to Rubik's cube solving which will later form an introduction to the machine learning approach in the dissertation.

Weeks 8 to 10 : 8th December - 29th December

Use the Christmas holiday to perform the bulk of the development of the neural net and run training sets based on inverted X-move scrambles to eventually have a program which can solve a Rubik's cube from X number of moves away.

Milestones: Search based code should now be working and after this section the neural net code should also be performing as expected.

Weeks 10 to 13 : 29th December - 19th January

Perform evaluation of the machine learning approach in terms of how it performs on solving a Rubik's cube based on the number of moves it is away from solved. Choose appropriate criteria to measure and record results thoroughly. Also perform analysis of the performance of the machine learning approach versus the search based method.

Milestones: In-depth, likely graphical, evaluation of how increasing the number of moves affects the performance of the machine learning approach, with particular reference to the following criteria: whether the cube is solved, number of moves to solve, time to generate solution algorithm. Similar analysis of the search based method compared with the machine learning when presented with X-move scrambles.

Weeks 14 and 15 : 19th January - 2nd February

By this point the majority of the code should be written and a lot of results will also have been gathered. I plan to use this period to refactor any of the code which may require reconsideration, as well as assessing the state of the project and which extensions I could feasibly achieve. I would also look to begin writing some of the dissertation especially the Introduction and Preparation chapters. As well as laying the groundwork for the Implementation chapter.

Milestones: A completed Introduction and Preparation chapter.

Weeks 16 and 17 : 2nd February - 16th February

Implement the extension of the complex machine-friendly algorithms into the search based approach (Thistlethwaite and Kociemba) and evaluate their results against those from the other search based options.

Milestones: Additional functionality added to the search based approach and analysis of the results completed.

Weeks 18 and 19 : 16th February - 2nd March

Use a machine learning approach to tackle the early part of the solve. Use a reinforcement learning technique and methods such as Roux (based on grouping like-coloured cells together) to develop a decent exploration heuristic. If this can not be made to perform well then look at plugging the search based solution into the machine learning approach, where the search-based solution brings the cube to a state where the neural net can solve it. There are explicit solving techniques which bring a cube to a nearly solved state which could be incorporated into this hybrid approach. If the machine learning approach can be made to work on the early parts of the solve then form a hybrid with the two different machine learning approaches.

Milestones: A hybrid of either machine learning - machine learning, or search based - machine learning which can solve a fully scrambled cube.

Weeks 20 and 21 : 2nd March - 16th March

Evaluate the performance of the new hybrid with the pure search-based method with full scrambles now as opposed to the earlier evaluation with X-move scrambles.

Milestones: Detailed graphical analysis of the performance of the hybrid with the search-based in terms of: whether the cube is solved, time taken to generate solution algorithm and total number of moves in the solution algorithm.

Weeks 22 and 23 : 16th March - 30th March

Implement a graphical representation of the simulation developed at the start. If this extension is deemed unattainable then incorporate an open-source visual representation of a Rubik's cube in order to display the programs solving the cube to a user.

Milestones: A graphical Rubik's cube which can be scrambled using input and solved using one of the programmed techniques.

Weeks 24 to 27 : 30th March - 27th April

Use the rest of the Easter holiday to revise for exams and write up the remaining aspects of the dissertation. Notably finish the implementation chapter and the evaluation chapter. Also review the previous chapters and the state of the project as a whole. Begin to draw final conclusions.

Milestones: Complete chapters 1-4 of the dissertation.

Weeks 27 and 29 : 27th April - 11th May

Meet with supervisor at the very start of term and discuss conclusions and the overall evaluation of the project before finishing these chapters and any appendices which are not already finished. Aim to finish the dissertation and submit a week early to allow full focus on exams after this point.

Milestones: Complete dissertation

Week 30 : 11th May - 18th May

A week of leeway to allow for any unexpected events towards the end as well as any final editing and formatting problems.

Milestone: Submission of Dissertation.

Appendix B

Additional Results

B.1 Tests for Normality

I attempted repeatedly to perform a chi-squared test for normality on my data sets in order to show they closely resemble normal distributions. However, because of the unusual characteristic of the data – at least for CFOP – only consisting of odd, or even, solution lengths – depending on the length of the scramble – this meant that the chi-squared tests would reject the hypothesis despite the data seeming to resemble a normal distribution very closely. Despite repeated attempts to address this problem using both Matlab and R, I was unable to come up with a satisfactory solution where the scrambles could pass the chi-squared test. Moreover, even when I did almost manage to, the sheer size of the data set (1,000,000 data points) meant that some of the slight irregularities in the data would throw off the test for normality significantly. Instead I decided to plot, using Matlab, a normalised histogram representing the length of solutions, as well as a normal distribution centred on the mean of the data points, and with the same variance – this is represented by a red line in the following rough graphs. By inspection it was easy to ascertain at which point the results resembled a normal distribution.

B.1.1 CFOP

At 2-move scrambles the results do not seem to be very close to resembling a normal distribution:

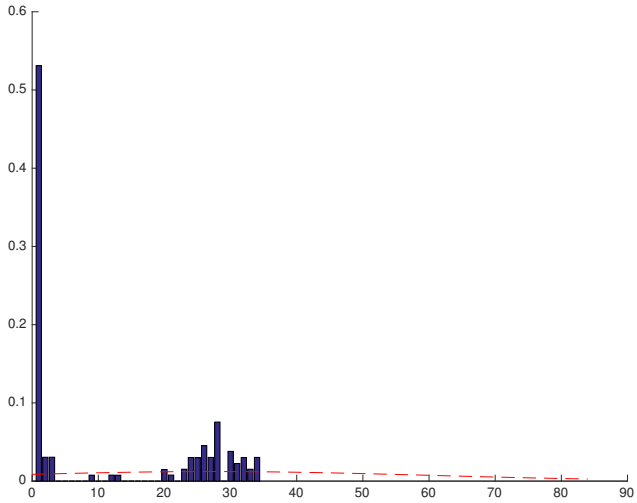

Figure 52: Normality of fit for CFOP 2-move scramble solutions.

Even by 4-move scrambles the results were beginning to resemble normal distributions:

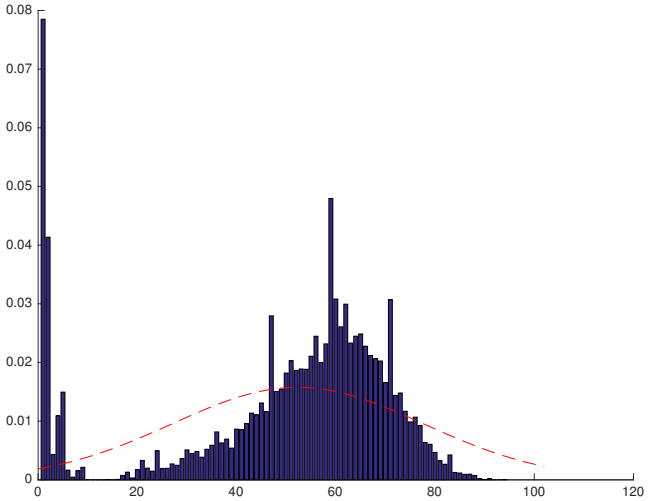

Figure 53: Normality of fit for CFOP 4-move scramble solutions.

By 8-move scrambles the results are significantly close that I am happy to say that they resemble an approximate distribution, despite the minority surge near the origin:

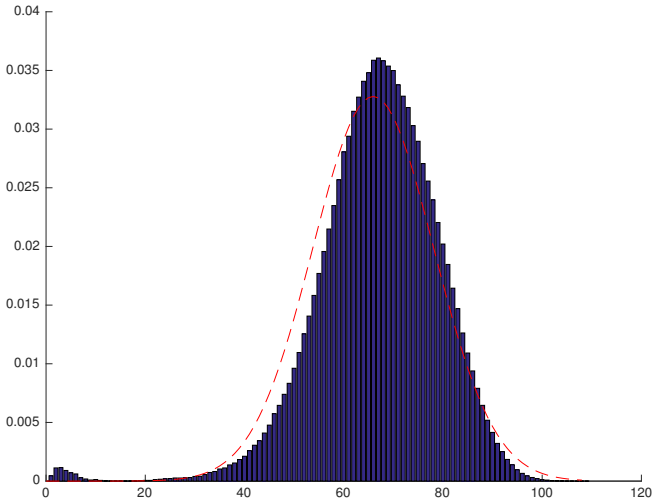

Figure 54: Normality of fit for CFOP 8-move scramble solutions.

From this point on the distributions get closer and closer until they fall almost perfectly within the normal distribution curve. So I chose 8 as the lower bound for the CFOP solutions.

B.1.2 Fridrich

The graphs for Fridrich showed an almost identical trend and from 8-move scrambles onwards they resembled normal distributions. For this reason I chose 8 as the cut-off for comparing between Fridrich and CFOP.

B.1.3 Ortega

B.2 Significance Tests

Using the same t-statistics as detailed in the Evaluation chapter :

$$t = \frac{\mu_C - \mu_F}{\sqrt{\left[\frac{\left(\sum C^2 - \frac{(\sum C)^2}{N_C}\right)}{N_C(N_C-1)}\right] + \left[\frac{\left(\sum F^2 - \frac{(\sum F)^2}{N_F}\right)}{N_F(N_F-1)}\right]}}$$

I carried out significance tests at an alpha level of 0.0005 – any value greater than 3.291 again indicates significance – to test the following hypotheses – t values are again given to 3 significant figures:

From the results in the previous section of this appendix I had found that the Ortega method followed a normal distribution after 6 moves, as CFOP and Fridrich only followed normal distributions from 8 moves away I calculated t-values from 8-move scrambles.

The Ortega method solves the cube in significantly fewer moves than the CFOP method – every value is significant, confirming the hypothesis:

Moves in Scramble	t-value
8	1130
9	1240
10	1330
11	1380
12	1420
13	1450
14	1470
15	1480
16	1490
17	1500
18	1510
19	1510
20	1520

The hypothesis for this t-test was: the Fridrich method solves the cube in significantly fewer moves than the Ortega method – every value is significant, confirming the hypothesis:

Moves in Scramble	t-value
8	1190
9	1230
10	1190
11	1230
12	1260
13	1280
14	1290
15	1300
16	1300
17	1300
18	1310
19	1320
20	1310