AALBORG UNIVERSITY

The Faculties of Engineering, Science and Medicine
Software
Strandvejen 12-14
Telephone 96 35 97 31
Fax 98 13 63 93
http://tnb.aau.dk

Title:
Rubik's Cube

Theme:
Network and Algorithms

Project period:
P2, spring semester 2010

Project group:
A215

Participants:

Alex Bondo Andersen

Mikkel Bach Klovnhøj

Mikael Midtgaard

Jens Mohr Mortensen

Dan Stenholt Møller

Rasmus Veiergang Prentow

Christoffer Ilsø Vinther

Advisors:
Anders Bruun
Heather Baca-Greif

Page count: 100

Appendices count: 4

Finished: 27/5–2010

Synopsis:

The goal of this project is to get a solid understanding of the Rubik's Cube and create an application which can solve the Rubik's Cube. In this report the origin of the Rubik's Cube will be presented along with some similar puzzles. In order to get a better understanding of the Rubik's Cube, its construction will be described along with some general terminology. Graph and group theory will be described along with an explanation of how these theories can be applied to the Rubik's Cube. Because there are many ways to solve a scrambled Rubik's Cube, there have been chosen two very different solving algorithms. Their functionality is described along with a description of how these have been implemented. Finally the difference between these two algorithms will be discussed along with how the implementation of them can be improved.

Contents

Preface

This report is written by seven software engineer students attending Aalborg University, group A215, and is associated with a P2 project in 2010. The main theme of this semester is "Network and Algorithms" and our project deals with the Rubik's Cube. The course of the project commenced on February 1, 2010, and the paper was handed in on May 27, 2010.

Readers are required to have a basic understanding of programming and mathematics in order to properly understand the contents of the report.

The report is divided into four parts and an appendix; Prologue, Theory, Implementation and Epilogue. Every chapter except for the preface and problem definition starts with a head and ends with a tail. They act as an introduction and a summary. Heads and tails are written in italic and are separated from the rest of the report by lines. Additionally we have produced an application, which is described further in the report and appended on a CD-rom.

All citations are written in square brackets such as [xx]. The number is a reference to the source, which can be found in the bibliography on page 98. Rubik's Cube is a registered trademark, but the trademark symbol is omitted during the report.

Group theory related names, such as moves, are written in **bold** and variable and function names are written in *italic*.

We, the authors, would like to use this opportunity to thank our main advisor and contextual advisor for their assistance in finding credible sources, and guidance, and generally aiding in the process of writing this report.

We would also like to thank Leif Kjær Jørgensen and Diego Ruano Benito for assisting us in obtaining usable sources concerning group theory. Furthermore we would like to thank Herbert Kociemba for his response to the e-mail we sent him.

Part I

Prologue

1 Introduction

Since 1977, when the Rubik's Cube was initially released for sale, the Rubik's Cube has frustrated, inspired and entertained many people. This 3x3x3 cube has so many possible settings that the solution can not just be guessed out of sheer luck. Because of this a community around solving the Rubik's Cube has emerged. The community is divided into two groups both concerning efficient solving – one efficient time-wise and the other efficient twist-wise i.e. solving in the least amount of time and solving in the least number of twists [13].

The group concerning time-wise efficiency – often referred to as speedcubing – is the largest group of the community and the majority of the competitions held by the WCA[1] [26] revolve around speedcubing.

The first official competition was held in 1982 in Hungary and is regarded the first World championship. Since 2002 there have been held annual world championships and plenty other events concerning speedcubing.

The group of the community concerning twist-wise efficiency is much smaller than the speedcubing group. The majority of the research in the twist-wise efficient area is published as scientific articles explaining different solving algorithms. Even though competitions with the goal of the least amount of twists to solve the Rubik's Cube are held, many of the twist-wise efficient algorithms are not useful for human solving. These algorithms rely on computer power to look through a large amount of possibilities, which is not a viable option for a human competitor.

The ultimate goal for the twist-wise efficiency community is to find *God's algorithm*, which is the algorithm that solves the Rubik's Cube in the absolute least number of twists from any given position. A part of finding *God's algorithm* is to find the number of twists needed to perform it. The upper bound of the Rubik's Cube is the proven number of twists needed to solve the Rubik's Cube from any position. The lower bound is the least number of twists required to solve the Rubik's Cube from the currently known worst case position.

[1] WCA, World Cube Association, is the official organization for Rubik's Cube related competitions.

2 Problem Definition

2.1 Problem Analysis

The group concerned with the twist-wise efficiency of the Rubik's Cube attempts to prove the upper and lower bounds of the Rubik's Cube. An important algorithm in this group is Kociemba's optimal solver, which is the twist-wise most efficient solver [16]. In the speedcubing group the time-wise efficiency is a primary concern and beginner's algorithm is a foundation for solving the Rubik's Cube in this group.

It would be interesting to test these two algorithms both in time-wise and twist-wise efficiency, so a comparison between the two algorithms can be made. This is interesting because we were unable to find any previous comparisons of the two algorithms.

Another interesting part of the twist-wise concerned group is finding *God's algorithm*. Since *God's algorithm* is unique for every position the length of the solution also very interesting, or namely the longest length. This length is where the upper and lower bounds meet.

Hereby it is interesting to examine how the upper and lower bounds have been proven and how the process of proving them has progressed.

2.2 Problem Statement

The following problem statement has been defined:

> *How have the upper and lower bounds of the Rubik's Cube progressed and how have they been proven?*
>
> *How efficient is Kociemba's optimal solver compared to beginner's algorithm and how can this be tested?*

2.3 Problem Limitations

We will only look at the improvements of the upper bound since 1981, when Thistlewaite published his proof of the upper bound of 52 [9].

The efficiencies we wish to test are the twist-wise efficiency – the fewer twists, the more efficient – and the time-wise efficiency, which is the amount of time it takes the algorithm to find a solution. Both of these will be based on a computer solving the Rubik's Cube since this will give the most reliable results.

3 Recreational Mathematics

The Rubik's Cube is related to other recreational mathematical puzzles, which have inspired to the creation of the Rubik's Cube. To better understand the Rubik's Cube this chapter presents its roots. This chapter presents a definition of recreational mathematics and a few examples of recreational mathematical puzzles related to the Rubik's Cube.

3.1 Definition

Recreation means to do something which is amusing or relaxing. Mathematics is somewhat harder to give a precise definition of due to the vast amount of subjects that fall under this term. For the purpose of defining recreational mathematics we define mathematics as: "*Science of structure, order, and relation that has evolved from counting, measuring, and describing the shapes of objects.*"[3]

Recreational mathematics is hereby defined as problems, puzzles or games involving counting, measuring, or use of shapes which are fun and interesting to laymen as well as mathematicians. [22] [25, p. 18]

3.2 Puzzles

This project is dedicated to the Rubik's Cube and the cube will be covered in detail later in this report. This section will instead describe some recreational mathematical puzzles related to the Rubik's Cube.

3.2.1 Magic Square

A Magic Square is a square, which is divided into a number of sub squares. The number of sub squares in any row or column is referred to as the "order" of the Magic Square. In each sub square there is a positive integer. In order for the Magic Square to be "magic" the sum of any row, column, and diagonal must be the same, this sum is referred to as the magic constant. See table 3.1 below.

The Magic Square [1] originates from ancient China. It was said that the people near the river Lo made offerings. Every time they made an offering a

6	1	8	15
7	5	3	15
2	9	4	15
15	15	15	15

Table 3.1: *A Magic Square of the order 3, by adding the three numbers in any row, column or diagonal, the magic constant is seen to be 15*

tortoise emerged from the river. On the back of the tortoise there was said to be a Magic Square.

The Magic Square from this tale was of the order 3. This is not the only order in which a Magic Square can be created; it is possible to make a Magic Square of the order n. Generally a Magic Square of the order n contains the numbers from 1 to n^2. It has been proven that it is not possible to make a second order Magic Square since simply trying all possible squares with the numbers 1, 2, 3 and 4 inside results in no combination that gives the same sum on each row, column, and diagonal.

In order to solve the Magic Square, it is needed to know the magic constant – the constant which every row, column and diagonal adds up to for the given order n. This constant can be computed with formula 3.1.

$$M(n) = \frac{n \cdot (n^2 + 1)}{2} \tag{3.1}$$

A Magic Cube is created from squares put on top of each other so they make up a cube form. This makes it clear that there is a connection between Magic Squares and Magic Cubes. The following section will demonstrate how the Magic Cube is related to the Rubik's Cube and thereby show that the Magic Square is related to the Rubik's Cube as well. An example of the Magic Cube can be seen on figure 3.1.

7	11	24
23	9	10
12	22	8

(a) *Top layer.*

15	25	2
1	14	27
26	3	13

(b) *Middle layer.*

20	6	16
18	19	5
4	17	21

(c) *Buttom layer.*

Figure 3.1: *This is a Magic Cube split up into 3 magic squares.*

3.2.2 Magic Cube

Both a Magic Square and a Magic Cube have a magic constant, which is the sum of each row, column, and pillar. However this is where the similarity ends.

It has been shown how to calculate the magic constant in a Magic Square. In a Magic Cube there is not a big difference in the formula to calculate the

magic constant.

$$M(n) = \frac{n \cdot (n^3 + 1)}{2} \tag{3.2}$$

As shown in the formula the only difference is the power of n that is changed from 2 to 3.

To create a Magic Cube, there are some parts that need to be explained. These parts can be seen on figure 3.2.

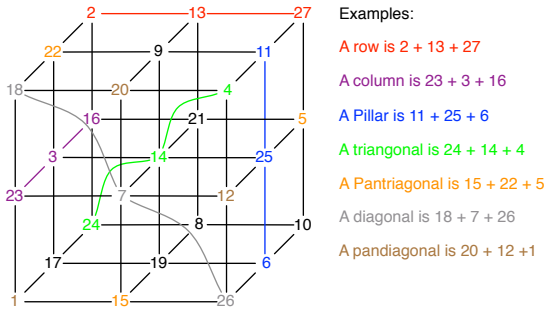


Figure 3.2: *This is a Magic Cube where the colors show all of the parts.*

Because of all these different parts there are a lot of different ways to define the Magic Cubes. The simplest of them is called a simple Magic Cube. The only requirements to make such a cube is the following:

- All 9 rows, columns, and pillars must be equal to the magic constant.
- All 4 triagonals must also be equal to the magic constant.

The Rubik's Cube has quite a few similarities with the Magic Cube. If all the integers in the Magic Cube was replaced with the small cubes of the Rubik's Cube called cubies, a Rubik's Cube would emerge. There are three differences. The first is that the Magic Cube consists of numbers whereas the Rubik's Cube has colors, which are different on each face. The other difference is that the Magic Cube has a number in the center where the Rubik's Cube center is invisible and not of importance. The last difference is that when permuting the Rubik's Cube, it is necessary to move several cubies as opposed to the Magic Cube where integers can be changed individually. See figure 3.3.

This Magic Cube should not be confused with the later presented Magic Cube made by Ernö Rubik.

Figure 3.3: *A Rubik's Cube.*

3.2.3 Magic Puzzle

The Magic Puzzle is also known as the 15-puzzle [12, pp. 48-50]. It is a puzzle that consists of a tray with 15 square tiles and an empty square arranged in a 4x4 contraption.

It has never been determined who actually invented the Magic Puzzle, but Samuel Loyd who was an American chess player and puzzle author claimed that he invented the Magic Puzzle and therefore he got the credit. This claim was rejected by a research of Jerry Slocum. He discovered that there was a wooden version of the puzzle already in 1865 that was manufactured by the Embossing Co. Jerry Slocum searched for the patent and found it, US 50.608, and this patent was applied for by Henry May.

Jerry Slocum also found a patent by Ernest U. Kinsey that was published August 20th 1878. This version by Ernest U. Kinsey was a 6x6 version of the puzzle which also prevented the tiles from being lifted out.

Permutations

The tiles in a Magic Puzzle can in theory be arranged in 16! different positions [20]. This limit can however not be reached because a permutation is needed to switch the tiles. The permutation must be an even or odd number of transpositions depending on where the position of the empty square is.

The tiles are often numbered or labeled with small pictures which – when assembled correctly – form a larger picture. This is related to the Rubik's Cube because having a solved picture in a Magic Puzzle is like having a solved face in the Rubik's Cube.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(a) *Figure of Magic Puzzle.*

15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	

(b) *Figure of Magic Puzzle with inverse numbers.*

Figure 3.4: *Illustrations of legal and illegal permutations of the Magic Puzzle.*

For instance we got the figure 3.4a and want to switch the tiles to be positioned like on figure 3.4b [12, pp. 48-50]. This permutation requires an odd transposition of the seven pairs (1,15), (2,14), (3,13), (4,12), (5,11), (6,10), and (7,9). This permutation is not possible because it requires an even number of transpositions to get the empty square at the same position. If we color the contraption like a chess board (see figure 3.5) we can see that every odd transposition makes the empty square change color and with every even transposition the empty square lands on a square of the same color.

Therefore the number of different positions is $\frac{16!}{2}$. If the empty square has to be in a fixed position the possible permutations will be $\frac{15!}{2}$. These permuta-

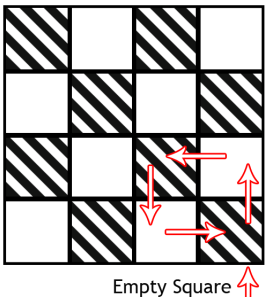


Figure 3.5: *Figure of a Magic Puzzle with an Empty Square.*

tions are almost like the ones the Rubik's Cube uses and they actually inspired Ernö Rubik in his creation of the Rubik's Cube [19, pp. 7-9].

This chapter gives a definition of recreational mathematics and shows three puzzles, which are all related to the Rubik's Cube; Magic Square, Magic Cube and Magic Puzzle. The Magic Square is the predecessor to the Magic Cube, which is the predecessor to the Rubik's Cube. The permutation from the Magic Puzzle inspired the creation of the Rubik's Cube, which uses a similar principle for moving the cubies around.

4 Origin of the Rubik's Cube

In this chapter we will describe the history behind Ernö Rubik and how he got the idea for the Rubik's Cube in order to get at better understanding of it. Furthermore the legal issues and the problematics in regards to the patenting of the Rubik's Cube will be described. The purpose of this chapter is to give a contextual perspective on the Rubik's Cube.

4.1 Ernö Rubik

Ernö Rubik is the inventor of the world famous Rubik's Cube. He was born in Budapest, Hungary in 1944. His father was a flight engineer and his mother was a poet. He graduated from the Technical University in Budapest as an architectural engineer. After his graduation he stayed at the university to teach interior design.

In January 1975 Rubik applied for a patent for his invention in Hungary that was originally made to help his students. Two years later in 1977 he got the patent on the Magic Cube. This Magic Cube is actually the same cube as todays Rubik's Cube, he just named it differently. His Magic Cube should not be confused with the Magic Cube described in section 3.2.2.

In the 1980's he became a professor and started the Rubik Studio, which employs a dozen people to design furniture and toys. Since the studios' opening Rubik has produced several other toys, including Rubik's Snake. Most recently the studio began developing computer games. He also became the president of the Hungarian Engineering Academy in 1990. The same year he created the International Rubik Foundation to support especially talented young engineers and industrial designers.

4.2 Rubik's Cube

In the 1970's when Ernö Rubik was teaching at the university he wanted to create a tool that would help his students to better understand tree-dimensional design. He wanted a design with blocks that could be moved individually, but also able to move several blocks at a time. Initially he attempted to do this with a cube held together by rubber bands. This failed. He then concluded

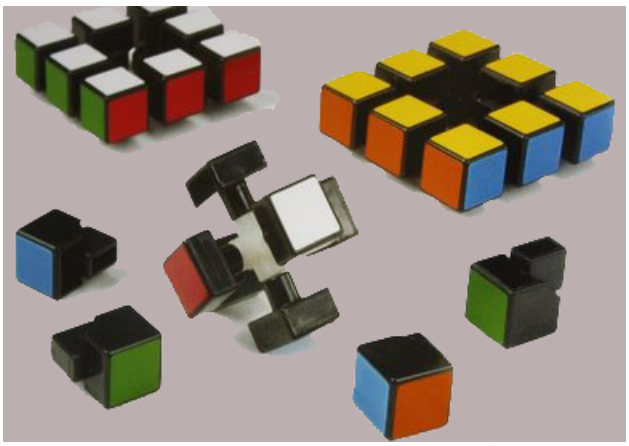


Figure 4.1: *Figure of a disassembled Rubik's Cube.*

after looking at a Magic Puzzle (see subsection 3.2.3) that the pieces must hold each other in place. Thereby he created what was then called the Magic Cube.

In the end of 1970's a Hungarian Businessman showed the Magic Cube at the Nuremberg toy fair and made it popular in Europe. The company Ideal Toy bought exclusive rights for the Magic Cube, but changed the name of the cube to Rubik's Cube within a year in order to get trademark protection.

At that time there were also two others applying for patent for products similar to the Rubik's Cube. One of them was an American man named Doctor Larry D. Nichols and his cube was a 2x2x2 cube, which was held together with magnets. See section 4.3. The other one who applied for patent was a Japanese man named Terutoshi Ishige. He applied for patent a year after Ernö Rubik. Terutoshi Ishige's cube was almost identical to the Rubik's Cube.

Ideal Toy Company was bought by CBS Toy Company in 1982 and the trademark passed with it, but they sold the rights to Rubik's Cube to Seven Towns which is a toy company in Great Britain, and they are still producing the Rubik's Cube today.

4.3 The Nichols Cube Puzzle

Dr. Larry D. Nichols studied chemistry at DePauw University in Greencastle, Indiana, before moving to Massachusetts to attend Harvard Graduate School. He is a lifelong puzzle enthusiast and inventor who began developing a twist cube puzzle with six colored faces in 1957. It was made of eight smaller cubes assembled to a 2x2x2 cube. The eight cubes were held together by magnets.

On April 11, 1972 he was granted U.S. Patent 3,655,201 on behalf of Moleculon Research Corp. U.S. Patent 3,655,201 covered Nichols Cube and the possibility for making larger versions later. This was two years before Ernö Rubik took out the patent for his Rubik's Cube in Hungary.

In 1982 Moleculon Research corp. sued Ideal Toy Company that had the U.S. Patent 4,378,116 for Rubik's Cube because they believed that Ideal Toy Company violated their patent, but the U.S. District Court ruled in Ideal Toy

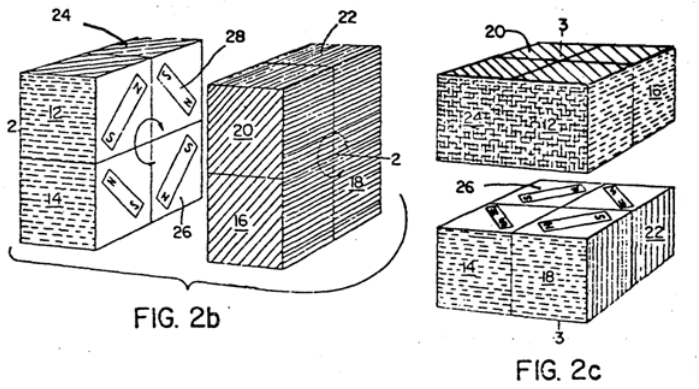


Figure 4.2: *Figure of Nichols Patent.*

Company's favor. In 1986 the Court of Appeals ruled that the Pocket Rubik's Cube 2x2x2 was guilty of infringement but not the 3x3x3 Rubik's Cube.

This chapter describes how Ernö Rubik got the idea for the Rubik's Cube. It is also stated that Ernö Rubik was not the only one with an idea of a cube shaped puzzle. This chapter gives a better understanding and a contextual perspective of the creation of the Rubik's Cube.

5 The Upper and Lower Bounds

In this chapter the process of proving the upper and lower bounds will be described. The upper and lower bounds are, as their evolution, essential to this project.

The lower bound is the number of twists required to solve the Rubik's Cube in the position which requires the most twists to solve. The upper bound is the lowest number of twists proven to solve a Rubik's Cube in any position.

The different bounds throughout the history can be seen on figure 5.1. The upper bound closes in on the lower bound and at some point the two bounds will merge to one [16], which is the length of the longest solution of *God's algorithm*.

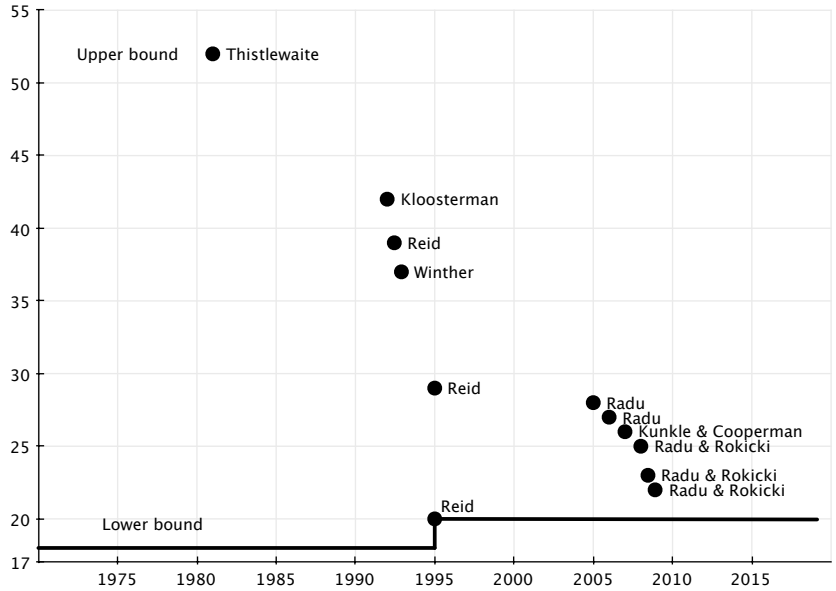


Figure 5.1: *The current upper and lower bounds. The x-axis represents time and the y-axis represents moves.*

5.1 The Current and Previous Upper Bounds

The progression of finding and proving the current and previous upper bounds has been a long process. This is due to the vast amount of different positions a Rubik's Cube can be in. This have inspired a small group of people to dedicate a lot of time to create and improve algorithms to solve arbitrary Rubik's Cubes.

Proofs of the upper bound has been published several times. A major breakthrough was when Thistlethwaite's algorithm was proven to be able to solve an arbitrary Rubik's Cube in 52 twists or less. His algorithm was based on group theory where it goes through four subgroups [21]. This upper bound existed in over 10 years and the next upper bound was found in 1992 by Hans Kloosterman, which reduced the upper bound to 42 moves [17, p. 44]. He modified Thistlethwaite's algorithm and found some shortcuts to reduce the moves by replacing one subgroup with a different subgroup and removing a move between the third and fourth subgroup.

In May 1992, Michael Reid reduced the upper bound to 39 moves by using a three subgroup algorithm and a thorough analysis of the subgroups [17, p. 52].

One day after Michael Reid lowered the bound, Dik T. Winter reduced the upper bound to 37 moves. He did it by analyzing a subgroup of Kociemba's algorithm and combining this with Kloosterman's algorithm [17, p. 53].

In January 1995 Michael Reid took the lead in lowering the upper bound. By using parts of Kociemba's algorithm he lowered it from 37 to 29 moves [17, p. 55].

In December 2005 Silviu Radu reduced the upper bound to 28 moves by extending Michael Reids work and again in April 2006, he lowered it to 27 moves [17, p. 58].

In August 2007 the upper bound was lowered to 26 by Kunkle and Cooperman [17, p. 63].

In 2008 Tomas Rokicki and Silviu Radu lowered the upper bound in the following months using their set solver, which is described in chapter 10 [17, p. 66]:

- March: lowered the bound to 25 moves by using only home PCs.
- April: lowered the bound to 23 moves by using idle time on the Sony Pictures Imageworks computer farm.
- August: Using more idle time on the computer farm to lower the bound to 22 moves.

To prove the current upper bound they needed to compute 1,265,326 different sets. At the moment they have not proven that the upper bound is 21, but the computer farm is currently working on it. They expect that it is possible to lower the upper bound to 20. This means that an arbitrary Rubik's Cube could be solved in just 20 moves.

5.2 The Lower Bound

As mentioned previously in this chapter the lower bound is the number of twists required to solve the Rubik's Cube in the position which requires the most twists to solve. For now it is proven that the superflip position (see figure 5.2), is the position which requires the most twists to be solved [14]. The shortest path from the superflip to the solved position is 20 twists [16]. This has been proven by trying every possible solution with 19 twists or less. There has not been found a position that requires more than 20 twists, which makes the lower bound 20 twists.

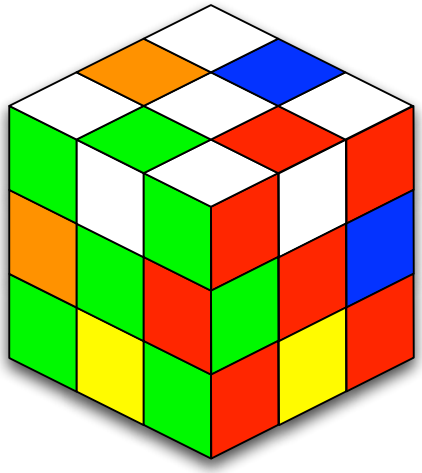


Figure 5.2: *The superflip – every edge is flipped. The lenght of the optimal solution to this position is 20 twists. A sequence to this position is:* ***F B U2 R F2 R2 B2 U' D F U2 R' L' U B2 D R2 U B2 U***

This chapter describes the history of the upper and lower bounds and gives an explanation of the two.

Part II

Theory

6 Terminology

In order to fully understand the theoretical part of this report it is necessary to know the used terminology.

6.1 General Cube Terminology

- **Center**: Center cubies have one facelet and are placed at the center of each face and are immovable unless the cube is turned.
- **Cuber**: The self reference for people who are devoted to the community of the Rubik's Cube.
- **Cubicle**: An imaginary static frame that the cubies can be placed in.
- **Cubie**: The cubie is a small piece of the Rubik's Cube, which has one, two, or three facelets. The Rubik's Cube consists of 26 cubies.
- **Corner**: Corner cubies have three facelets and are placed at the corners. The Rubik's Cube has eight corners.
- **Edge**: Edge cubies have two facelets and are placed at the edges of each face. The Rubik's Cube has 12 edges.
- **Face**: A face is an entire side of the cube. The Rubik's Cube has six faces.
- **Facelet**: The small stickers on the cube. Each face has nine facelets.
- **Move sequence**: The same as a sequence of twists.
- **Turn**: A turn of the cube is equal to rotating the whole cube 90 degrees i..e. changing the view angle.
- **Twist/move**: A rotation of a face.

6.2 Movement Notation Terminology

A Rubik's Cube consists of six faces and the notations of these are the following.

- **Front face – F**: This face faces the cuber.
- **Left face – L**: This face faces the left hand side of the cuber.
- **Right face – R**: This face faces the right hand side of the cuber.
- **Up face – U**: This face faces up.
- **Down face – D**: This face faces down.
- **Back face – B**: This face faces away from the cuber.
- **General move – M**: Can be any of the above faces.
- **Half-twist metric**: In this metric of notation half-twists (**M2**) are allowed. This metric will be used in this report.
- **Quarter-twist metric**: In this metric half-twists are not allowed.

A face can be twisted in two directions – clockwise and counterclockwise. When twisting a face the direction is determined as if you were facing the face. A twist in the clockwise direction has the same name as the face. i.e. a clockwise turn of the right face is notated **R** and pronounced "right". A counterclockwise twist of the right face is notated **R'** and pronounced "right prime". This goes for all the faces. A 180 degree twist of a face is denoted **M2** e.g. **R2** will twist the right face 180 degrees, known as a half-twist. Beside the normal face twists there are middle twists. These are denoted **Mm Mm' Mm2** e.g. **Rm2** is a twist of the middle slice looking from the **R** face. This twist is equal to the two twists **R2 L2**. Middle twists will not be used unless stated.

A turn of the cube can be done in six directions. Clockwise and counterclockwise around each of the three axes.

6.3 Algorithm Terminology

- e: The solved state of the Rubik's Cube (unit cube).
- **S**: The 18 standard twists of the half-twist metric: **U U' U2 D D' D2 F F' F2 B B' B2 R R' R2 L L' L2**.
- s: A specific position of the Rubik's Cube.
- **H**: The set of positions which follows these rules:
  - Every cubie is correctly orientated (see section 12.2).
  - Every edge cubie not containing either an up facelet or a down facelet is positioned in the middle layer.
- **A**: A set of twists containing the following twists: **U U' U2 F2 R2 B2 L2 D D' D2**.

- **G**: All positions which can be obtained without disassembling the Rubik's Cube.
- $r(s)$: The relabeling of the position s.
- $\mathbf{S}^n$: Set of move sequences consisting of every possible sequence of maximum n moves in **S**.
- $\mathbf{S}^*$: Every combination of twists in **S**.

Note that positions can be considered as the shortest move sequence that transforms the Rubik's Cube from e to the given position s.

This chapter gives some basic terms used in this report.

7 Group Theory

In this chapter there will be explained some basic group theory and how it can be applied to the Rubik's Cube. This is needed because the solving algorithms are based on group theory. There will be a calculation to determine the number of different permutations of the Rubik's Cube, in order to determine the size of the Rubik's Cube group.

7.1 Permutations

In this section the number of permutations of the Rubik's Cube will be calculated [16]. The first corner cubie can be placed in one of eight cubicles, the next corner cubie can be placed in one of the seven remaining cubicles. This means that the corner cubies can be placed in $8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 8!$ positions. This does not mean they have the correct orientation since a corner cubie has three different facelets and therefore three different possible orientations. This means there are 3^8 orientations of the eight corner cubies, which yields $8! \cdot 3^8$ corner permutations.

The 12 edge cubies can be placed in 12 different cubicles and every edge cubie can be orientated in two different ways. This yields 2^{12} different orientations and 12! different positions of the edge cubies. This gives us $2^{12} \cdot 12!$ different edge permutations.

All of these corner and edge permutations adds up to a total of

$$3^8 \cdot 2^{12} \cdot 12! \cdot 8! = 519,024,039,293,878,272,000 \approx 5.2 \cdot 10^{20} \text{ permutations.}$$

There are however some limitations to a correctly assembled Rubik's Cube. Only seven out of the eight corner cubies can be orientated independently. The last corner cubie's orientation is directly dependent on the other corner cubies. The same logic applies to the edge cubies, which means that only 11 edge cubies can be orientated independently. Furthermore the orientation of the edge cubies is also dependent on the orientation of the corners, which halves the number of possible permutations.

The legitimate Rubik's Cube has

$$3^7 \cdot 2^{10} \cdot 12! \cdot 8! = 43,252,003,274,489,856,000 \approx 4.3 \cdot 10^{19} \text{ permutations.}$$

7.2 Definition of the Rubik's Cube Group

To understand how the Rubik's Cube can be described as a group, it is necessary to understand what a group is. In group theory a group is written as $(Set, Operator)$ where the set is the elements the group consists of and the operator is the operation that is performed between the elements. When looking at the Rubik's Cube as a group the set will be a move or a move sequence that is applied to the Rubik's Cube. The group is denoted $(\mathbf{G}, *)$. The operator in the Rubik's Cube group will be asterisk $(*)$ because the movements of the Rubik's Cube are not commutative. The moves are not commutative because it matters in which order they are applied to the Rubik's Cube [19, p. 157].

To prove that the Rubik's Cube is a group there are four points it must fulfill:

1. When two elements are combined with an operation the new element these create must also be a part of the group. For instance a move $\mathbf{M}_1$ and a move $\mathbf{M}_2$ then $\mathbf{M}_1{*}\mathbf{M}_2$ will form a new move or move sequence which is also a part of the group.

2. e is an empty move, which does not change the configuration of the Rubik's Cube. If the move $\mathbf{M}{*}e$ is performed, only the move $\mathbf{M}$ is performed (the move e could be a 360 degrees twist of a face), which means that $\mathbf{M}{*}e = \mathbf{M}$. The empty move e should not be confused with the solved state e.

3. If $\mathbf{M}$ is a move then there is a reverse move, which is called **M'**. Therefore $\mathbf{M}{*}\mathbf{M'} = e$, which means every element in the group has a reverse move. This holds for the Rubik's Cube since every move has a reverse, e.g. $\mathbf{R}$ has **R'**, and a move sequence can be reversed by first inverting the order and then reverting each move in the sequence.

4. The operation must be associative meaning that the parentheses can be reordered without affecting the result. A move sequence can be defined as the orientation and position it puts each cubie in. If c is a positioned cubie, $\mathbf{M}(c)$ will be the cubicle which c ends in after the move is applied. For instance the move $\mathbf{R}$ will move the *up-right* cubie to the *back-right* cubicle, so therefore $\mathbf{R}(\textit{up-right}) = \textit{back-right}$. If a move sequence is applied, then the operation will look like this $\mathbf{B'}(\mathbf{R}(\textit{up-right})) = \textit{down-back}$.
   $*$ is associative because $(\mathbf{M}_1{*}\mathbf{M}_2){*}\mathbf{M}_3 = \mathbf{M}_1{*}(\mathbf{M}_2{*}\mathbf{M}_3)$ for any moves $\mathbf{M}_1$, $\mathbf{M}_2$ and $\mathbf{M}_3$. $(\mathbf{M}_1{*}\mathbf{M}_2){*}\mathbf{M}_3$ and $\mathbf{M}_1{*}(\mathbf{M}_2{*}\mathbf{M}_3)$ does the same operation to every cubie. This is the same as saying $[(\mathbf{M}_1{*}\mathbf{M}_2){*}\mathbf{M}_3](c) = [\mathbf{M}_1{*}(\mathbf{M}_2{*}\mathbf{M}_3)](c) = \mathbf{M}_3(\mathbf{M}_2(\mathbf{M}_1(c)))$ for any cubie c.

In fact it is possible to define the Rubik's Cube only using the following set of moves: (**U D F B L R**) since every other move and move sequence can be made out of these six moves. This is called the Singmaster notation [8, p. 7]. To make shorter move sequences two of such twists will be denoted with the letter of the move appended by **2**, three of such twists will be denoted with the letter of the move appended by a prime (**'**) as stated in section 6.2.

As an extension to the first point some moves will form a new move rather than a move sequence, e.g. $\mathbf{R}{*}\mathbf{R} = \mathbf{R2}$. This goes for any move sequence only containing moves of the same face.

7.3 Subgroup

In this section it will be proven that a subset of a group is also a subgroup. A nonempty subset $\mathbf{H}$ of the group $(\mathbf{G}, *)$ is called a subgroup of $\mathbf{G}$ if $(\mathbf{H}, *)$ is a group.

Let $(\mathbf{G}, *)$ be a group. A nonempty subset $\mathbf{H}$ of $\mathbf{G}$ is a subgroup of $(\mathbf{G}, *)$ if, for every $a, b \in \mathbf{H}$ then $a * b^{-1} \in \mathbf{H}$.

Proof. First, suppose $\mathbf{H}$ is a subgroup. If $b \in \mathbf{H}$, then $b^{-1} \in \mathbf{H}$ since $(\mathbf{H}, *)$ is a group. So, if $a \in \mathbf{H}$ as well, then $a * b^{-1} \in \mathbf{H}$.

Conversely, suppose that, for every $a, b \in \mathbf{H}$ then $a * b^{-1} \in \mathbf{H}$.

- First, notice that $*$ is associative since $(\mathbf{G}, *)$ is a group.
- Let $a \in \mathbf{H}$. Then, $e = a * a^{-1}$, so $e \in \mathbf{H}$.
- Let $b \in \mathbf{H}$. Then $b^{-1} = e * b^{-1} \in \mathbf{H}$, so inverse exist in $\mathbf{H}$.
- Let $a, b \in \mathbf{H}$. By the previous step, $b^{-1} \in \mathbf{H}$, so $a * (b^{-1})^{-1} = a * b \in \mathbf{H}$. Thus, $\mathbf{H}$ is closed under $*$.

Therefore, $(\mathbf{H}, *)$ is a group, which means that $\mathbf{H}$ is a subgroup of $\mathbf{G}$. □

In this chapter the Rubik's Cube has been introduced as a group. The operations that can be applied to the Rubik's Cube group have been explained. It has furthermore been proven that a subset of a group is a subgroup, as long as the stated conditions are met, which means that the same theory that applies to a group also applies to a subgroup.

8

Graph Theory

This chapter concerns the graph theory in relation to the Rubik's Cube. The shortest path of a graph and the calculation of the diameter of a graph will be described. This can be used as a method to prove how many twists are needed to solve the Rubik's Cube from any position, also called the upper bound. The chapter will end with an example using a much smaller Rubik's Cube graph; the middle movement graph.

A graph consists of a set of vertices [18, p. 592], and a set of edges. There are several types of graphs, but in this chapter only simple graphs are described. A vertex can be connected to other vertices by edges. An edge is a line between two vertices (an edge can in some graphs connect a vertex to it self, called a loop, but such graphs are not taken into consideration here). Figure 8.1 illustrates the basic principles of a graph.

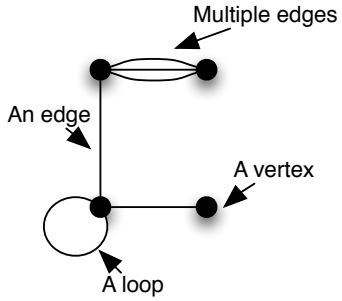
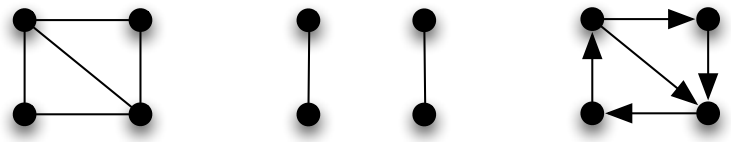


Figure 8.1: *This is a graph with multiple lines and a loop. The graph is connected, but not simple.*

If there is no more than one edge between any two vertices in a graph, then the graph is said to be simple. A graph is said to be connected if it is possible to start in an arbitrary vertex and then reach every other vertex by traveling along the edges of the graph. Figure 8.2b illustrates an unconnected graph while figure 8.2a illustrates a connected graph.

Graphs can be directed. This means that the edges has a direction and travel can only be done along an edge in one direction. Figure 8.2c illustrates a directional graph.

A graph can be weighted, which means that every edge will have a weight. This weight can represent different things, distance between vertices, price for traveling between vertices or what ever makes sense for the given weighted graph.



(a) *A simple connected graph.* (b) *A simple unconnected graph.* (c) *A simple directed graph*

Figure 8.2: *Two simple graphs and one simple directed graph.*

Graphs can be used to illustrate many different things such as different relationships, geographical maps, and tournaments [18, pp. 592-593].

For a graph of the Rubik's Cube it would be ideal to make vertices represent positions and edges represent twists. The weight of the edges would then be the number of twists required to get from one vertex of the edge to the other – hence the weight for every edge would be 1 if only edges representing a single twist is included in the graph.

The Rubik's Cube graph can have different sizes depending on the allowed twists, e.g. a graph only considering **Rm2 Um2 Fm2** twists can be made into a relatively small graph (see figure 8.4 for illustration) compared to the full Rubik's Cube graph, which contains $4.33 \cdot 10^{19}$ vertices.

8.1 Shortest Path

The shortest path from one vertex to another in a graph can be found by checking the length of each possible path. This is easy for a small graph such as the middle movement graph, see section 8.3.1, which will be described later in this chapter. For a bigger graph such as the full Rubik's Cube graph this is practically impossible.

The shortest path between two vertices can be found by using an algorithm e.g. *Dijkstra's Algorithm* [18, p. 651]. The description of the algorithm is omitted for brevity. *Dijkstra's Algorithm* takes an weighted graph and two vertices as input. Because of this the Rubik's Cube graph has to be weighted. Since each twist contributes to the total number of twists by the same number, one, each edge is given the weight one.

8.2 Solving the Diameter

To find the diameter of a graph is to find the longest shortest path between any two vertices in the given graph i.e. the shortest path of the given graph with the highest value. This can be done by using e.g. *Dijkstra's Algorithm* on every pair of vertices in the graph. By doing so, every shortest path is found between

every pair of vertices. Then the highest value of these is the diameter of the given graph.

In figure 8.3 there are 10 vertex pairs, note that the shortest path from e.g. A to B is the same as B to A. The shortest paths of this graph is: A to B: 69 ; A to C: 48 ; A to D: 63 ; A to E: 61 ; B to C: 21 ; B to D: 36 ; B to E: 34 ; C to D: 15 ; C to E: 13 ; D to E: 28. The diameter of this graph is 69, since this is the highest value among shortest paths of the graph.

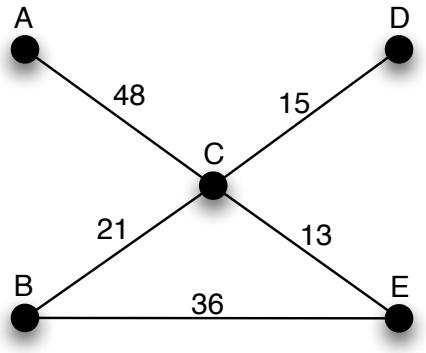


Figure 8.3: *A weighted graph.*

8.3 Describing the Cube as a Graph

In order to describe the Rubik's Cube as a graph it is necessary to determine the edges and the vertices. For the Rubik's Cube graph edges are defined to represent twists and vertices to represent the positions. The full Rubik's Cube graph is indeed a simple graph. A move can be reversed which means that the graph is undirected. In order to get from a position a to an adjacent position b one can get there by a single move. The full Rubik's Cube graph consists of approximately $4.33 \cdot 10^{19}$ vertices – all having 18 edges. It is practically impossible to draw this graph. Therefore the graph will be explained with a much simpler graph; the middle movement graph.

8.3.1 The Middle Movement Graph

This graph is a Rubik's Cube graph consisting of only the moves that twist the middle sections. This is **Rm2 Fm2 Um2**. Note: **Rm2** = **R2 L2**. This graph is fairly small, since it only consists of eight vertices and 12 edges. See figure 8.4 [19, pp. 158-167].

Because of the relatively small size of this graph the computation of the diameter is a relativly simple task. It is possible to calculate the distance from all vertices to all other vertices, but since the graph is clearly symmetrical many calculations can be omitted. It is easy to see that the diameter must go from one vertex to the vertex furthest away e.g. from the solved state to the *pons asinorum*[1]. The diameter of this graph is three.

[1]Pons asinorum is obtained from the solved state of a Rubik's Cube by the move sequence **Rm2 Fm2 Um2.**

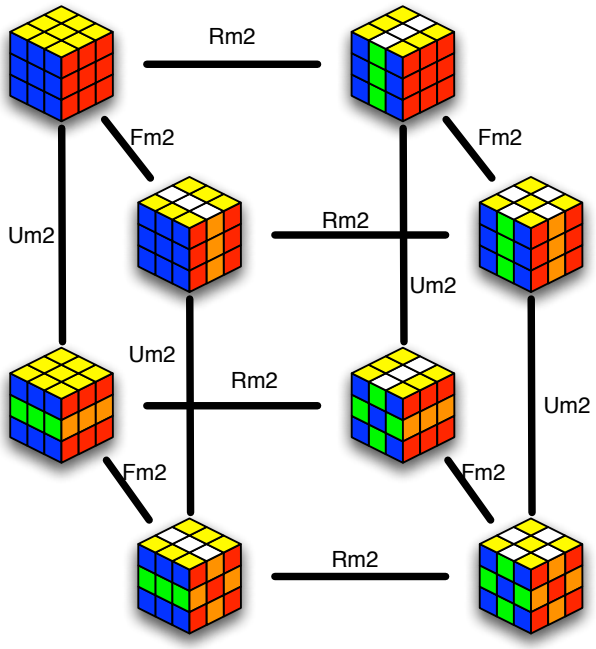


Figure 8.4: *The graph of the middle movement positions.*

A general description of graph theory has been presented in this chapter along with a way to calculate the diameter of a graph. For better understanding the theory, an example of applying the theory on the Rubik's Cube has been shown.

9 Rubik's Cube Solving Algorithms

This chapter will present two different algorithms, which can be used for solving a Rubik's Cube. The two algorithms presented in this chapter are beginner's algorithm and Kociemba's optimal solver. These algorithms will be used as a foundation for the implementation part.

9.1 Beginner's Algorithm

This section describes the easiest algorithm to remember for a human solver. The algorithm is divided into five different steps. Once the algorithm has been memorized, it is easy to recognize which step of the algorithm has been reached, so the correct move sequence can be applied. Beginner's algorithm can however be made more efficient by remembering another rather similar move sequence for a few of the steps. Despite of this beginner's algorithm is twist-wise quite inefficient, because the purpose of the move sequences is to reach the next step instead of reaching the solved state and thereby taking a solving detour.

9.1.1 Step 1 - Getting the Cross

The first step of beginner's algorithm [4] is to get a cross on any face. Getting a cross on a face means to align the facelets next to the center facelet, so that all of the aligned facelets are of the same color. At the same time the used edge cubies should have the same color of the center facelets on each of the two faces on which they are on. See figure 9.1 for a correctly assembled cross on the green face.

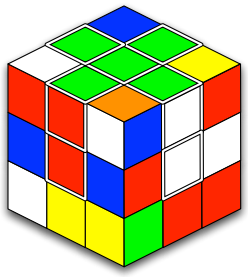


Figure 9.1: *A first layer cross on the green face*

The face on which the cross is being assembled is set to be the up face. An edge cubie that consists of the same colors as the center cubie of the up face and the center cubie of the front face is placed in the bottom of the front face. With two twists of the front face the edge cubie is positioned correctly in the cross. If the edge cubie

is orientated correctly the Rubik's Cube is turned and the process is repeated until the cross is assembled. However if the cubie is orientated in the wrong way the following move sequence will change its orientation without ruining any part of the cross that may already be assembled:

F' U L' U' or **F U' R U**

9.1.2 Step 2 - Completing the First Layer

When the cross is completed the next step is to position the corner cubies of the first layer correctly. The first layer is set as the down face. A corner with a facelet of the color of the down face is positioned directly above its correct position. The correct position is between the three faces that have the same colors as the three facelets of the corner cubie.

Once the cubie is above its correct position, the Rubik's Cube should be viewed in such an angle that the cubie is in the upper right corner of the front face, the following move sequence is repeated until the corner cubie is orientated and positioned correctly (see figure 9.2):

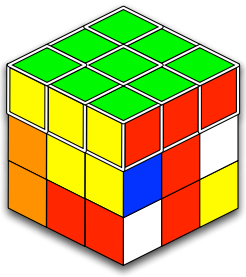


Figure 9.2: *The first layer completed. Note that the first layer is up.*

R' D' R D

If the cubie is above the correct position the algorithm twists the corner clockwise and positions it in the correct position. If the cubie is in the correct position the algorithm positions the cubie above the correct position. The maximum number of repetitions until the cubie is orientated and positioned correctly is five, because the cubie can be two twists away from its correct orientation. The move sequence can be performed inverted which twists the corner counterclockwise and looks as follows:

D' R' D R

If the correct move sequence is used the maximum number of repetitions is three. If number of twists and time used is not of importance it is only necessary to remember one of them.

9.1.3 Step 3 - Completing the Second Layer

Now the Rubik's Cube is turned, so the first layer will be down. The purpose of this step is to position the four edge cubies belonging to the second layer correctly. The move sequence used in this step either swaps the top edge cubie of the front face with the left or right edge cubie. The edge cubie that needs to be positioned must have a facelet that is of the same color as the front face. This facelet must be on the front face. The Rubik's Cube needs to be turned to get the correct front face, and then the up face is twisted in order to get the correct cubie on the front face. There is however a difference in which of the

two faces is used as the front face, because if the wrong face is used as front face the cubie will only be positioned correctly but not orientated correctly. That facelet of the edge towards the front face must be of the same color as the front face. If the edge cubie is to be swapped with the right edge cubie of the front face the following move sequence is used:

U R U' R' U' F' U F

If the cubie is to be swapped with the left edge cubie of the front face the following move sequence is used:

U' L' U L U F U' F'

If none of the edge cubies who belong to the second layer are in the third layer and the first two layers are still not completed. This occurs if the edges are all in the second layer but not positioned or orientated correctly. One of the move sequences can be used to swap an edge cubie from the third layer with one of the edges in the second layer which is not positioned or orientated correctly. Now the edge cubie belonging to the second layer can be moved to its correct position. See figure 9.3.

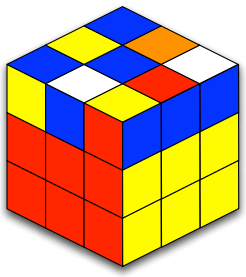


Figure 9.3: *The first two layers completed*

9.1.4 Step 4 - Getting the Last Layer Cross

Solving the last layer is divided into four substeps. The order of these steps can be different and still yield the same result. We start out by getting the cross in the last layer (see figure 9.4c), which is the same as the cross on the first layer. The move sequence used is however different. The only move sequence used in this step is the following:

F R U R' U' F'

Besides remembering the move sequence it is also important to know how the Rubik's Cube should be turned.

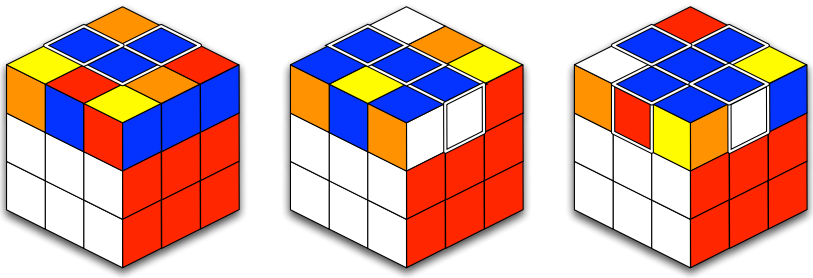
If the up face colors form a line (see figure 9.4b). The Rubik's Cube can be turned until the line forms a horizontal line. If the move sequence then is performed, the cross will be formed.

If the up face colors form an opposite "L" (see figure 9.4a) with the up face colors the move sequence will form the line. The Rubik's Cube must be orientated with the opposite "L" in the back left corner of the Rubik's Cube.

If the up face colors do not form the opposite "L" the line or the cross (see figure 9.3), the move sequence will form the opposite "L".

To orientate the cross correctly there is one move sequence to remember:

R U R' U R U2 R'



(a) *The up face colors form the opposite L.* (b) *The up face colors form the horizontal line.* (c) *The up face colors form the cross.*

Figure 9.4: *The steps in the completion of the last layer cross*

Again it is important to know how to turn the Rubik's Cube. By twisting the upper layer it is possible to position the upper layer so it either has two edge pieces next to each other or directly across from each other.

If the correct edge cubies are next to each other. The Rubik's Cube must be turned so it has a correct edge cubie on the back face and a complete face on the right face. The move sequence will then make it possible to twist the up layer so that all the edge pieces are correctly positioned and orientated (see figure 9.5).

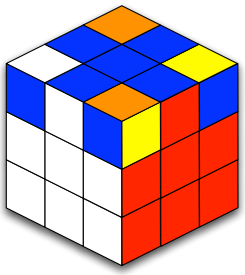


Figure 9.5: *The correct orientated last layer cross.*

If the correct edge pieces are across each other the move sequence can be performed a single time to get two edge pieces next to each other.

9.1.5 Step 5 - Completing the Last Layer

The purpose of this step is to position and orientate the corners correctly. Firstly the corners must be positioned correctly. To do so there are two move sequences to remember – one for rotating three corners clockwise and one for rotating them counterclockwise. It is however only necessary to remember one of them and repeat that one until the corners are positioned correctly, if the number of twists is unimportant to the solver.

U R U' L' U R' U' L
This move sequence will rotate the corners counterclockwise.

U' L' U R U' L U R'
This move sequence will rotate the corners clockwise.

The orientation is again important to position the corners. If one of the corners already is positioned correctly. That corner is chosen not to be moved.

If the three other corners need to be moved counterclockwise. The correct corner is positioned as the front right corner. The move sequence will then position the corners correctly.

If the three other corners need to be moved clockwise. The correct corner is positioned as the front left corner. The move sequence will then position the corners correctly.

If there are no correct corners. One of the two move sequences above performed once will yield a correctly positioned corner cubie.

To orientate the corners correctly the move sequence from orientating the corners in the first layer is used:

R' D' R D

Instead of turning the whole Rubik's Cube to the next corner, the Rubik's Cube is locked on one face. When going to solve the next corner cubie the upper layer is twisted until the incorrect corner is positioned at the front-right corner cubicle of the Rubik's Cube and the move sequence is repeated. The Rubik's Cube is now solved as seen on figure 9.6.

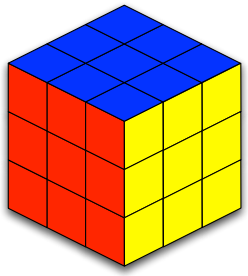


Figure 9.6: *The solved cube – e*

9.2 Kociemba's Optimal Solver

Kociemba's optimal solver is an algorithm created with the purpose of finding the twist-wise optimal solution to any scrambled Rubik's Cube [11] [24]. It has been developed by Herbert Kociemba who has studied physics and mathematics at Technische Universität Darmstadt [6] (1974-1979) and is currently teaching at a gymnasium. His interest in the Rubik's Cube started in the 1980 and he implemented his first algorithm – Kociemba's Two Phase algorithm – in 1990. See appendix A for the e-mail correspondence with Herbert Kociemba.

This section uses the following terminology:

- a: A position in $\mathbf{G}$.
- b: Path from a to a position in $\mathbf{H}$ using move sequences in $\mathbf{S}^*$.
- c: Path from b to e using move sequences in $\mathbf{A}^*$.
- ab: Combination of the two move sequences a and b, in that order.
- d: Distance of phase 1 .
- $d2$: Table lookup for the distance from ab to e.

Kociemba's optimal solver consists of two phases and is based on Kociemba's Two Phase algorithm. The first phase is based on a principle called relabeling, which will be defined at first.

9.2.1 Relabeling

The relabeling process starts with choosing an up face with a corresponding down face. Then choosing a front face with a corresponding back face. Each facelet with the color of the up or down face is marked with the letters "UD". Every edge cubie that is not labeled with "UD" is labeled with "FB" on the front and back face. Figure 9.7 shows an example. When relabeling a Rubik's

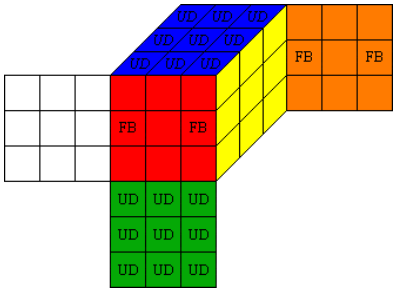
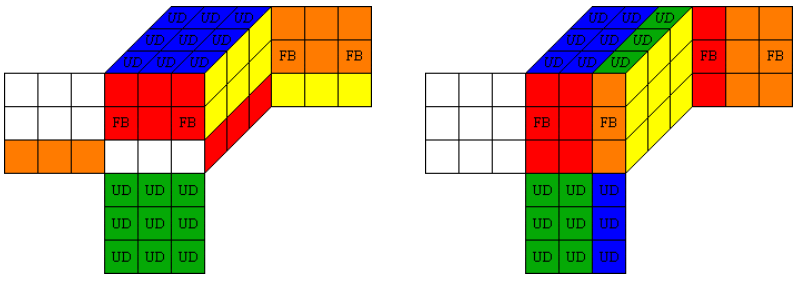


Figure 9.7: *A relabeled Rubik's Cube with the up and down faces as blue and green and red and orange as front and back.*

Cube in the position s it is written as $r(s)$. A Rubik's Cube in any position, s, is said to be in the set of positions called **H** (see subsection 9.2.2) if and only if $r(s) = r(e)$.

9.2.2 The Subgroup H

The moves **U**, **U'**, **U2**, **D**, **D'**, **D2**, **R2**, **L2**, **F2** and **B2** is the set of moves **A**. Using moves from **A** on a position in **H**, will always result in a Rubik's Cube in **H**. The reason for this can easily be tested with a Rubik's Cube. If using one of the three up face moves or one of the three down face moves, the "FB" labels are not moved and the "UD" labels are simply rotated on the face that is being twisted, see figure 9.8a. The last four moves are all very similar to each other. If any side face – not up or down – is twisted 180 degrees, the three facelets on the up face is moved to the down face and vice versa and thereby keeping all the "UD" labels on the up and down face and keeping the orientation correct. The two remaining relabeled facelets are swapped which keeps the "FB" labels placed and orientated correctly, see figure 9.8b.



(a) *A relabeled cube that has been permutated with the move **D**.*

(b) *A relabeled cube that has been permutated with the move **R2**.*

Figure 9.8: *Two positions which have been permuted with a move in **A**.*

9.2.3 Overall Description

The first phase takes a scrambled cube a and relabels it $r(a)$ then it finds a move sequence b which will transform the relabeled cube into the subgroup $\mathbf{H}$. This move will be denoted: $r(a) * b \in \mathbf{H}$. The second phase will then determine the length from the position ab to the unit position e by a table lookup.

Algorithm 1 Kociemba's Optimal Solver [16]

```
1: d = 0
2: l = ∞
3: while d < l do
4:    for b ∈ S^d do
5:       if r(ab) ∈ H then
6:          if d + d2(ab) < l then
7:             l = d + d2(ab)
8:          end if
9:       end if
10:   end for
11:   d = d + 1
12: end while
13: l is now the length of the optimal solution to a
```

The search distance in the algorithm d (see figure 9.9) is initially set to zero and the total length l is set to infinite. The total length is the amount of moves used to get from a to e, while d is the limited search length to find a move sequence that transforms the Rubik's Cube into a position in $\mathbf{H}$. The **while** loop will run as long as d is less than l.

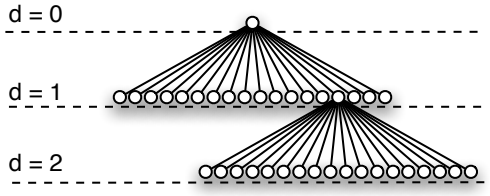


Figure 9.9: *As the distance of the search d increases, the possible move sequences expands exponentially. For each vertex on the graph there is 18 child vertices. The amount of leaves for each search depth would be* 18^d.

The **for** loop runs through all move sequences in the range d – recall that $\mathbf{S}^d$ is the set containing every move sequence that uses d twists. This search of moves in $\mathbf{S}^d$ is the first phase and is further described in subsection 9.2.4. If the move sequence transforms the Rubik's Cube into the subgroup $\mathbf{H}$, the algorithm checks whether $d + d_2(ab)$ is less than the length of the last solution. If so, $d + d_2(ab)$ is the new shortest move sequence, l, to e. The lookup table denoted $d_2(ab)$ returns the numbers of twist required to transform a position in $\mathbf{H}$ to the position e. This is done only by using moves in $\mathbf{A}$. The lookup table is the second phase and is further described in subsection 9.2.5. The first time a solution is found the length of this will be set to l and the solution is saved.

The **while** loop will end when d is incremented to l, then the optimal solution has the length l. The actual solution is b plus some move sequence in $\mathbf{A}^{d_2(ab)}$. Figure 9.10 illustrates the algorithm.

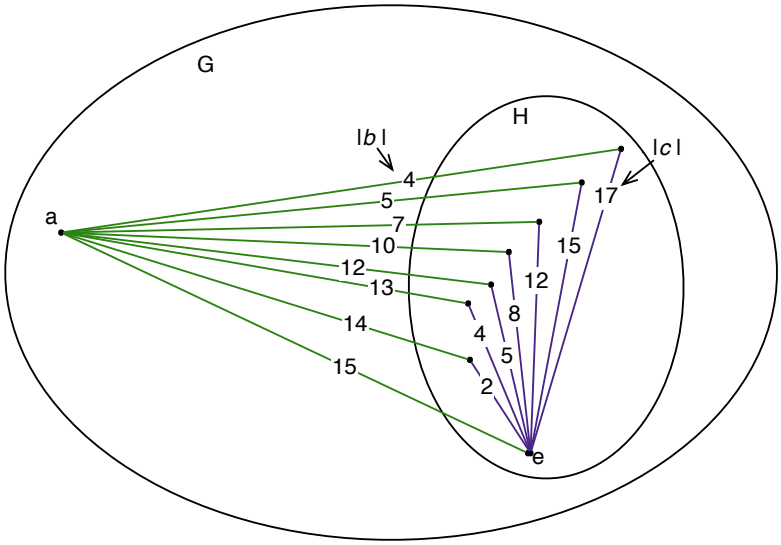


Figure 9.10: *The lines going from a to a point in* ***H*** *is the move sequence denoted b. The lines from points in* ***H*** *to the point e is the move sequence denoted c. The numbers on the lines are the number of moves. Note that the moves in c is decreasing as the numbers of moves in b is increasing. The line that goes directly from a to e is the shortest move sequence possible.*

9.2.4 First Phase

The first phase finds a move sequence, b, from a position a that transforms the Rubik's Cube into the subgroup **H**. This is done by going through all possible moves with a sequence of the length d. This is a breadth-first search algorithm [18, pp. 729-731]. In figure 9.9 the amount of elements in $\mathbf{S}^d$ to a specific d is illustrated. An example of a search from a position a is given. The algorithm starts by searching in $\mathbf{S}^d$, where $d = 0$. This will give the position a itself. The distance d is then incremented to one. This will result in 18 new possible move sequences all one twist away from a. Thereafter d is incremented and the algorithm now searches 18 moves from the previous positions obtained in the search where $d = 1$. This allows a possibility for optimization of the algorithm since some of the moves will eventually be the same. e.g. the two move sequences **R2** and **R' R'** (see section 7.2) is of length one and two respectively, but the result is the same transformed cube. The algorithm checks whether the transformed cube is in **H** by relabeling it and checking if $r(ab) = r(e)$.

d continues to be incremented until it reaches the length l. Since l starts with the value infinite, it has to be changed in order to stop the loop. This happens in the second phase.

9.2.5 Second Phase

The goal of the second phase is to find the length of the shortest move sequence to transform the Rubik's Cube from a position in **H** to e. A way to solve this problem is by having a lookup table. This table has to be very large, considering the amount of positions in **H**.

The amount of positions can be calculated by imagining that one were to assemble the Rubik's Cube, starting with a disassembled Rubik's Cube. The first corner cubie can be placed in eight different cubicles. The next corner has seven possible cubicles etc. Note that all cubies in a cube in **H** has the correct orientation. This gives $8! = 40320$ possibilities for the corners. The eight edges of the top and down layer can be placed similarly and also yields $8!$. The four edges of the middle layer can be placed in four different cubicles this yield $4! = 24$. Since it is impossible to swap two corners without swapping any edges the amount of possibilities is halved. The final result is

$$\frac{4! \cdot (8!)^2}{2} \approx 19.508 \cdot 10^9$$

elements in **H**. The actual implementation of such a table will be discussed in chapter 14.

In this chapter beginner's algorithm and Kociemba's optimal solver are described so they can be implemented.

10

Rokicki's Set Solver

The set solver was used to prove the current upper bound and therefore it is an important part of this project.

Tomas Rokicki has proved several upper bounds and got a Ph.D. in computer science in 1993 [16].

The algorithm used to prove the current upper bound (see section 5.1) is known as Rokicki's set solver and uses Kociemba's algorithm. The set solver is a viable method for proving the upper bound because it does not solve every single cube, but a whole set of cubes at a time, as the name suggests. This means that it solves approximately 19.5 billion cubes at a time. The set solver does this by finding all the move sequences of a relabeled cube of the distance d that transforms the cube into **H**. The set solver algorithm is described in pseudo code below in algorithm 2.

Algorithm 2 Set Solver [16]

```
1:  f = ∅
2:  d = 0
3:  loop
4:     if d ≤ m then
5:        for b ∈ S^d do
6:           if r(ab) = r(e) then
7:              f = f ∪ ab
8:           end if
9:        end for
10:    end if
11:    if f = H then
12:       return d
13:    end if
14:    d = d + 1
15:    f = f ∪ fA
16:    if f = H then
17:       return d
18:    end if
19: end loop
```

The set solver starts by initializing two variables **f** and d. **f** is a set that can hold all the positions of **H**. **f** is set to an empty set. The second variable is the distance d, which is the distance from a scrambled position a to a position in **H**. This distance d is initially set to zero.

Next the **while** loop is run. It will run until d is returned, which is when all positions in **H** has been found.

m is the maximum number of **S** moves performed from the position a. When d is equal to m only **A** moves are performed. If d is lower than or equal to m a for loop will be run. This loop performs all possible move sequences of the the length d, and adds the position ab to **f** if it is a position in **H**. For the sake of efficiency move sequences that give the exact same position more than once are not used. If a move sequence contained **F F'**, that part would of the move sequence would be unnecessary.

If **f** is equal to **H** all positions in **H** have been found. d will then be returned and the algorithm has finished. If this is not the case, d is incremented by one. The different **A** moves are performed on all the current **H** positions in **f** and the new **H** positions are saved in **f**. If **f** contains all positions in **H**, d is returned – if not the while loop continues.

When the algorithm has finished all the different possible **H** positions are saved, if the maximum distance m is set sufficiently high. The theoretically highest number of twists needed to transform any scrambled cube into **H** is 12. The set solver also performs moves that transforms a Rubik's Cube in **H** to a Rubik's Cube not in **H** and then back again by using **S** moves. This is done because more positions are found inside **H** when using **S** moves than when only using **A** moves [16].

In this chapter the set solver used to prove the current upper bound is described.

Part III

Implementation

11

Choices Prior to Implementation

This chapter presents our approach to test the two algorithms from chapter 9. Furthermore the choices prior to the implementation will be described.

We want to test the two algorithms from chapter 9. This can be done in two ways: Making the tests by hand or implementing the algorithms in an application, which can perform the tests. To test Kociemba's optimal solver and beginner's algorithm we will create an application that has a digitalized Rubik's Cube, which the user can permute and scramble. In this application the algorithms will be implemented for testing.

Before we start implementing our application we have to make some choices, namely: Programming language, version control, and which User Interface (UI) we want. These choices and the dilemmas, which we encountered are presented here.

- The first issue we will discuss is, which language we want to write our application in. The most important issue regarding the programming language is that all the members of the group have a knowledge of the language prior to the implementation phase. This leads us to choose Java as our programming language since we all have been following a course in this language.

- Secondly we have to choose a way to share our source code, since we want to work in smaller subgroups and then collect the work when we complete a task. For this we choose to use subversion (SVN) because we have all been using this protocol while working on the report and therefore know how to use it.

- We have to consider how to represent the Rubik's Cube in a UI. The choice is between Graphical User Interface (GUI) and Textual User Interface (TUI). We choose a GUI because it will be easier to see how the Rubik's Cube permutes in our application.

- Finally we have to decide how much time we want to spend on the GUI since we do not consider it as an important part of our project according to our problem limitations (see section 2.3). We choose to make a simple GUI with the Rubik's Cube to the left, buttons to right, and a console in

the bottom. The technical documentation of the creation of the GUI will be omitted.

This chapter presents the choices which we make before we start our actual implementation. What approach we will take to test Kociemba's optimal solver and the beginner's algorithm is described.

12

Digitalizing the Rubik's Cube

In order to implement solving algorithms to use on a Rubik's Cube the cube itself must first be digitalized. An interface must be created in order to see the result of the algorithms. In this chapter that process is described.

A Rubik's Cube is a rather simple three-dimensional structure, but implementing this structure into a computer system and getting it depicted on a two-dimensional screen is not a simple task. The Rubik's Cube is built up by 26 moving cubies held together by each other. This type of structure is not straight forward to implement in a desktop application, so a way to handle these cubies is needed.

A simple way to handle the Rubik's Cube in the program is a two-dimensional depiction and then to move the facelets around. The analogue to this on a real Rubik's Cube would be to take the colored stickers off and move them to their new position rather than actually move the cubies when twisting the Rubik's Cube. This approach would be far from reality and would make the implementation of solving algorithms, such as Kociemba's optimal solver and the beginner's algorithm more complicated, since they need to know the position of the cubies, and not just the facelets in order to determine the next step. A more object-oriented approach than to move the stickers would give a more useful structure for solving the problem. How this is done will be explained in the following section.

12.1 The Cube Structure

In this implementation we use an object-orientated approach to the problem. The Rubik's Cube is divided into its sub structures which consist of six faces each with nine shared cubicles each containing a cubie. There are three types of cubies and cubicles: corners, edges, and centers. The cubies either consist of one, two, or three facelets. On figure 12.1 the cube class diagram is shown and a detailed description of the classes will explained in the following subsections.

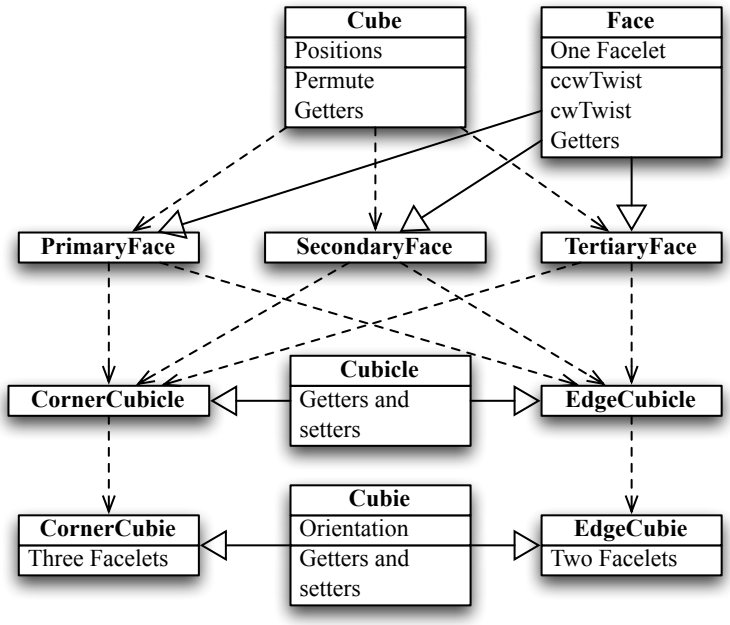


Figure 12.1: *Class diagram of the cube with properties and methods.*

12.1.1 The Cube

The cube class creates 20 cubicles in which the cubies can be positioned. Furthermore the cube class creates the faces and cubies. Even though the cube class initializes all objects, this is not drawn on the class diagram (see figure 12.1) because they are not used this way. The cube class also gives each cubie its facelets. The cubicles are then put into a face by its constructor. The cube class creates six faces – two primary, two secondary, and two tertiary faces. Each type of faces are placed opposite to each other. E.g. the up and down faces are both primary.

The cube has a static method, *permute*, it recieves two arguments, a cube object and a sequence of twists. The twist sequence is defined by the same class as the buttons in the GUI, namely the *MoveButtons* class. The moves are named by the Singmaster notation [8, p. 7] where the prime (') is replaces by a "P", e.g. **U'** is **UP**.

12.1.2 The Face

A face consists of nine cubicles which act as placeholders for the cubies. The center cubies will never move and therefore there is no reason to define a cubicle and a cubie for those. Instead the center piece of a face is granted a facelet, which defines the color of the face in the completed state of the Rubik's Cube. The face class constructor takes eight cubicles (four corners and four edges) and positions them in a clockwise order. In addition the face class implements two methods which twist the face clockwise or counterclockwise – namely *cwTwist* and *ccwTwist* respectively.

There are three different types of faces; primary faces, secondary faces, and tertiary faces, which are defined by their center facelet. Each type has its own subclass, which inherits from the face superclass. The reason for the different classes is that there is a difference in how the orientation of the cubies change depending on what type of face being twisted.

Each face is known by its type and a number 0 or 1 (see figure 12.2) e.g. the up face is know in the program as $PRIMARY\_0$. Its opposite is know as $PRIMARY\_1$.

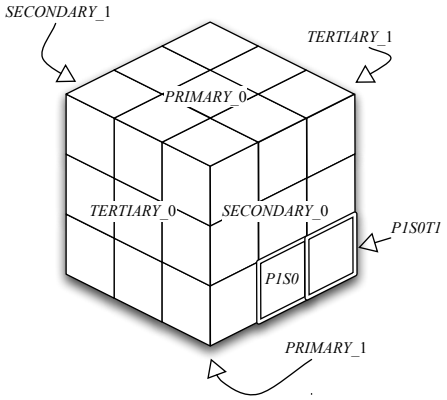


Figure 12.2: *The different faces of a Rubik's Cube. The colors can be defined as whatever one wants.*

12.1.3 The Cubicle

The cubicle class is a superclass and there are two subclasses to this class; corner- and edge cubicle. The corner and edge cubicle classes each have one instance variable, namely the cubie they hold. The cubie is set by the constructor of the cubicle class. The cubie can be replaced with another cubie with a setter.

The different positions are defined by the faces. There are two types of positions; corners and edges. An edge cubicle could be named $P1S0$; which means $PRIMARY\_1$ and $SECONDARY\_0$. See figure 12.2.

The corner to the right of $P1S0$ would be $P1S0T1$, which refers to the corner on the $PRIMARY\_1$, $SECONDARY\_0$, and $TERTIARY\_1$ faces. See figure 12.2.

12.1.4 The Cubie

The cubie class sets the orientation (see section 12.2) correctly by default, because the cube is assembled correctly when initialized. It implements methods that return the orientation and facelets of the cubie. The cubie class is a superclass and there are two subclasses to this; corner- and edge cubie. The corner cubies contain three facelets, where edge cubies only contain two facelets. The corner and edge cubie classes implements two different methods, which return the orientation of the cubie.

12.1.5 The Facelet

The facelets are defined in an enumeration which contains the colors of the facelets, it implements a *toColor* method which define the actual color of each facelet. The facelets are divided into primary, secondary, and tertiary facelets – each consisting of the facelets of two opposite faces.

12.2 Orientation

A face will move the cubies from one cubicle to another. See figure 12.3. By performing a twist the faces adjacent to the twisted face are also affected. E.g. when twisting the up face of a Rubik's Cube the cubies adjacent to the up face in R, L, B, and F faces will be permuted.

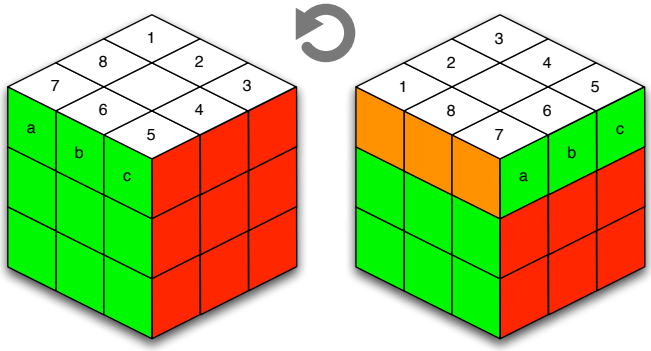


Figure 12.3: *A twist of a face will not only affect the twisted face but also its adjacent faces.*

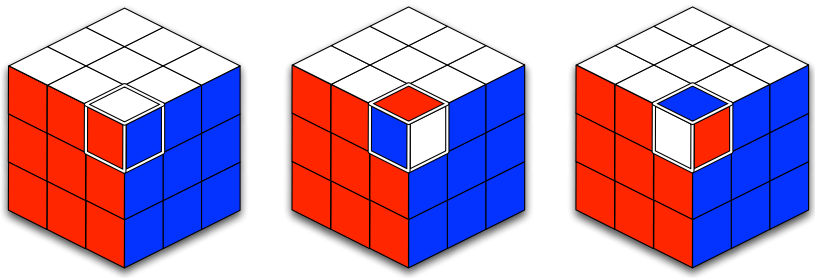
However the problem is that it requires variables to determine how the cubies are orientated. In order to draw a Rubik's Cube on the screen there are two things which are needed; the position of each cubie and the orientation of each cubie. This section will deal with the orientation of a cubie.

The way to define an edge cubie differs from the way to define a corner cubie, which is why these to are dealt with separately.

12.2.1 Corner Cubie

A corner cubie in a given cubicle can be orientated in this cubicle in three different ways. See figure 12.4 for the three different orientations of the white/blue/red corner cubie. Since each corner cubie has one of each facelet type – one primary, one secondary, and one tertiary – the orientation can be defined from where one of these facelets are positioned. These different orientations can thereby be defined with an integer between 0 and 2. 0 if the primary facelet of the cubie is on the primary face, 1 if the primary facelet is on the secondary face, and 2 if the primary facelet is on the tertiary face.

In order to make it easier to draw the Rubik's Cube a secondary orientation is added. This orientation is defined from which face the secondary facelet is on. This orientation is also defined from an integer between 0 and 2. 0 if the



(a) *The white/blue/red corner is correctly orientated, which gives it the primary orientation value 0.*

(b) *The white/blue/red corner is incorrectly orientated, the primary orientation value is 1 since the primary color is on a secondary face.*

(c) *The white/blue/red corner is incorrectly orientated, the orientation value is 2 since the primary color is on a tertiary face.*

Figure 12.4: *The three different orientations, which the white/blue/red cubie can have. In these figures the white and yellow faces are primary, the blue and green are secondary, and red and orange are tertiary.*

secondary facelet of the cubie is on the primary face, 1 if the secondary facelet is on the secondary face, and 2 if the secondary facelet is on the tertiary face. Note that a correctly orientated cubie will have primary orientation 0 and secondary orientation 1. It will also be easy to define a tertiary orientation of a cubie by where the tertiary facelet is positioned, but this will be redundant since this integer will always have the remaining value, e.g. the primary orientation is 0, the secondary orientation is 1, then the tertiary orientation must be 2. See figure 12.5 for a flowchart on how to find the primary and the secondary orientations of a corner cubie.

12.2.2 Edge Cubie

The orientation of an edge cubie can be defined with a boolean value; either it is correctly orientated or it is not. This is easiest to see when the cubie is in the right position. If the white/blue edge cubie is in its correct cubicle then it is correctly orientated if the white facelet is on the white face and the blue facelet is on the blue face. The cubie is wrongly orientated if the blue facelet is on the white face and the white facelet is on the blue face.

This raises the question: What is the orientation of a cubie which is not in its correct cubicle? This was not too difficult with corner cubies because they always had a facelet of each type, which an edge cubie does not. An edge cubie can be one of the following combinations of facelet types: Primary/secondary, primary/tertiary, or secondary/tertiary. With this knowledge it is possible to look at the "highest" facelet, meaning the facelet with the highest rank. The rank refers to the type of facelet, where primary is the highest, secondary the second highest rank, and tertiary the lowest. The orientation is then defined as the following: If the highest facelet of a cubie is on the highest face, then the cubie is correctly orientated. In order to stay in connection with corner cubie the correct orientation of a cubie is given the number 0 and the incorrect

orientation the number 1. Note that this is opposite to what is usually done in programming where 1 is true and 0 is false [23]. See the flowchart in figure 12.5.

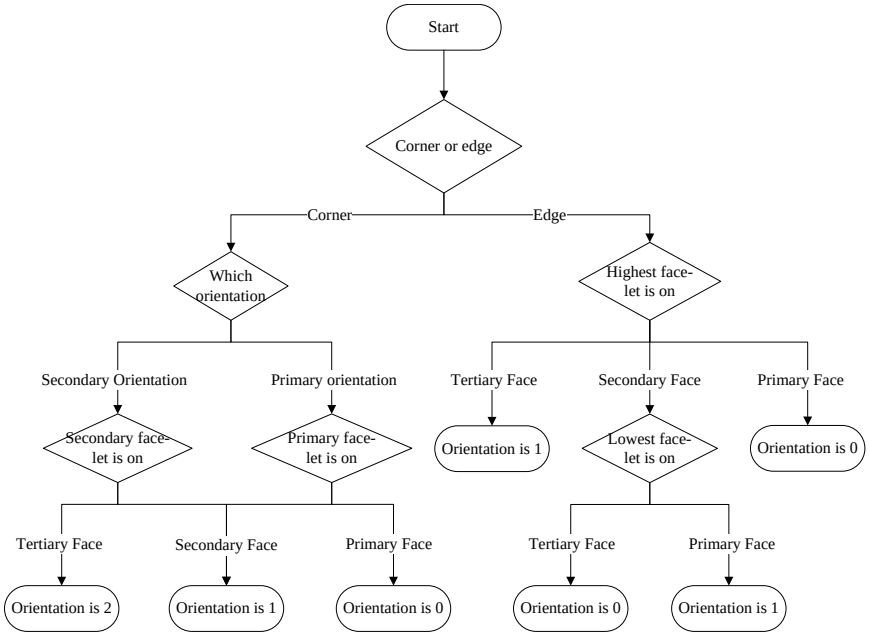


Figure 12.5: *This flowchart illustrates how to obtain the orientation of a cubie.*

In this chapter the process of digitalizing the Rubik's Cube has been described. This implementation gives a foundation for implementing the solving algorithms.

13

Beginner's Algorithm

In this chapter the process of implementing beginner's algorithm, which is described in section 9.1, will be presented. This implementation will not be as effective as beginner's algorithm is capable of since it is a proof of concept. How efficient it is will be presented in form of statistics about the number of twists it needs to solve the Rubik's Cube.

This chapter is divided into the different steps used in the beginner's algorithm. Each step is using different algorithms which all are generally structured in the same way.

For the sake of simplicity the first layer (down face) in our implementation will always be the same face. We have chosen that face to be the yellow face, the green is the front face, and the red is the right face. This will make the implementation process easier, but the solution will require more twists since it is unlikely that the yellow face is the optimal choice as the first layer face for every solve.

The implementation uses several algorithms which all are named algorithm XnN, where Xn is the step that the algorithm is used in and N is used to distinguish the algorithms from each other, which are used in the same step.

Most of them contains a **switch-case** statement which is dependent on some input specified for each algorithm. In all cases this is based on the point of view. Normally a human will simply rotate the whole Rubik's Cube and perform the same algorithm, but we found it easier simply to define the algorithms from the different locations.

13.1 The Steps

In the following the implementation of the steps of the beginner's algorithm will be described. The steps are based on the steps described in section 9.1.

13.1.1 Step 1 – Getting the Cross

This step will get the first layer cross, this equals to fitting and orientating of the edges in the yellow face. The step is generally described in subsection 9.1.1. First the method takes the edge cubie in cubicle $P1S0$ (see subsection 12.1.3) and checks if the edge cubie is positioned correctly. If that is the case and

the edge cubie is orientated correctly, the program proceeds to the next edge cubie. If the cubicle is positioned correctly, but not orientated correctly the method will perform an algorithm, which changes the edge cubie's orientation without ruining other possibly correctly positioned edge cubies. This algorithm is named $algorithm1A$. When the edge cubie is in the correct position and has the correct orientation, the program moves on to another edge cubie.

```
private void algorithm1B(EdgePos p){
  MoveButtons[] moves;
  switch (p) {
  case S0T0:
    moves = new MoveButtons[]{ F, U, FP};
    break;
  case S0T1:
    moves = new MoveButtons[]{ FP, U, F};
    break;
  case S1T0:
    moves = new MoveButtons[]{ BP, U, B};
    break;
  default:
    moves = new MoveButtons[]{ B, U, BP};
    break;
  }
  for(int i = 0; i < moves.length; i++){
    this.moves.add(moves[i]);
  }
  Cube.permute(cube, moves);
}
```

Code snippet 13.1: *This is algorithm 1B, which will move an edge piece from the middle layer to the top layer without ruining any edge pieces which are correctly positioned in the cross.*

If the edge cubie is not in its correct position, then the method checks where the cubie is positioned. Depending on which layer it is in, different actions will be taken:

1. If the edge cubie in question is in the first layer it is moved to the opposite layer.

2. If the edge cubie is positioned in the second layer an algorithm is performed, which moves the edge cubie to the last layer, which is the opposite of the layer in which we make the cross. This algorithm is called $algorithm1B$ (see code snippet 13.1). The algorithm takes a cubicle as input and performs a specific move sequence accordingly. The move sequences in the different cases are generally the same, but since we only view the Rubik's Cube from one angle, we have to move different faces in the move sequence accordingly to the cubicle which is input. e.g. if the cubicle is in the front and right faces the move sequence is performed as it originally is. However when the cubicle is in the right and back faces, every move will rotate 90 degrees, so that an **F** move becomes a **R** move etc.

3. If the edge cubie is in the last layer, no action is performed, since this is where we want it to be for now.

In the **for** loop, in code snippet 13.1, the moves are added to an array, which purpose is to write the moves in the GUI console. The *Cube.permute* method applies the moves to the cube.

Now that the edge cubie is in the last layer it needs to be moved to the position directly above where it is correctly positioned. The program does this by applying an **U** twist and checking if the edge cubie is above its correct position. The edge cubie is now positioned in the last layer face and the face which has the same color as the edge cubie's other facelet i.e. the facelet that is not yellow. In order to position the edge cubie correctly that face is twisted twice. The edge cubie is now in it's correct position, and if it is orientated correctly the program moves on to another edge cubie . If the edge cubie is oriented incorrectly $algorithm1A$ is performed. If more edge cubies need to be positioned or oriented correctly the program will continue with the same method until the cross is finished in which case the program moves on to the next step (see figure 13.1).

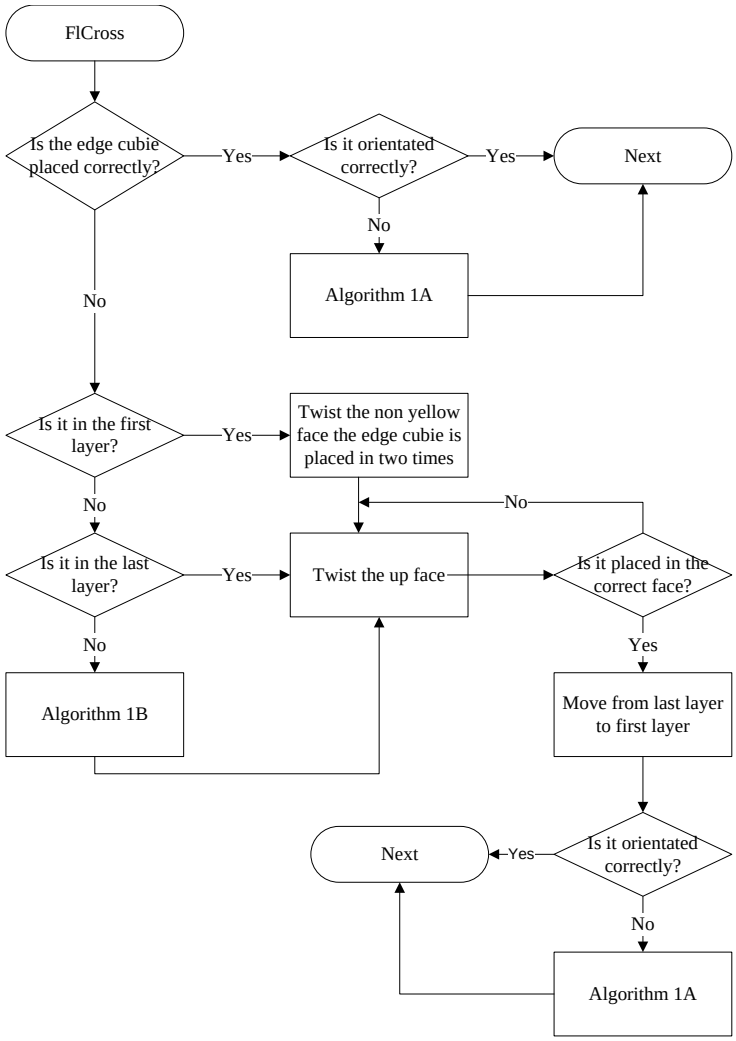


Figure 13.1: *The figure illustrates what is done in step 1.*

13.1.2 Step 2 – Completing the First Layer

This step positions the first layer corners into their correct position and orientation. The methods used for this step are very similar to the methods of step one. Like step one the algorithm starts with a specific cubie and then positions and orientates it correctly, before moving on to the next cubie This step is generally described in subsection 9.1.2.

First it checks whether the cubie is already in the correct position, if that is the case and it is not correctly orientated $algorithm2A$ will "rotate" this cubie in its cubicle until the orientation is correct without ruining other correctly positioned cubies.

If the cubie is somewhere else in the down layer it will be moved to the top layer by $algorithm2B$. By now we know that the corner cubie is either placed correctly or is in the white layer. The up face will be twisted until the cubie is placed directly above its correct place. Then $algorithm2B$ that was used to move the cubie up will now move it down. $Algorithm2A$ is then run until the cubie is orientated correctly.

13.1.3 Step 3 – Solving the Second Layer

In this step the second layer will be solved. This means that the edges in the second layer (middle layer) will be positioned and orientated correctly. In subsection 9.1.3 this step is generally described.

The method solves this layer by going through the four cubicles of the second layer and finding and placing the cubie in its cubicle

If the cubie is orientated incorrectly in the correct cubicle the program will use $algorithm3A$ to move the edge cubie to the top layer.

If the edge cubie is not in its correct position the program needs to know where the correct edge cubie is positioned. First the program checks for it in the top layer. If it is in the top layer **U** moves are applied to the Rubik's Cube until the edge cubie is positioned in such a way that $algorithm3B$ or $algorithm3C$ can be applied to move it to the correct position with the correct orientation. If the cubie is in the second layer but in a wrong cubicle, $algorithm3A$ will move it to the top layer. Then it can be positioned using **U** moves so that $algorithm3B$ or $algorithm3C$ can move it to its correct cubicle.

13.1.4 Step 4a – Getting the Last Layer Cross

In the previous steps the algorithms has solved one cubie at a time. This is not an option for this step. Here several cubies will be placed at a time.

This step will make the top face cross. Positioning the edge cubies correctly will not be done before the next step.

This step is performed by checking each case of possible orientations for the edge cubies in the top layer. There are three primary settings the top layer edge cubie can be in. See figure 9.4 in subsection 9.1.4 where this step is described generally. The three cases which the last layer can be within if the cross is not already present.

1. The white facelets of the top layer form an L-shape.

2. They form a row.

3. Only the center facelet is visible.

It is *algorithm*4*aA* that needs to be performed no matter what the case is. In case one the algorithm is applied twice, case two the algorithm is applied three times, case three the algorithm is applied once.

13.1.5 Step 4b – Completing the Last Layer Cross

In the theoretical description of the beginner's algorithm step five is the implementation's step five, six and seven. In this step the edges of the last layer cross will be positioned correctly. Their orientation will not be changed, since they are already orientated correctly. See subsection 9.1.5.

The four cubies in the white layer cross can only be in the four cubicles of the white face. This leaves us with essentially two cases. Either all cubies are placed correctly or two are placed correctly. This requires twisting of the up face until one of these cases appear.

The program only needs to respond to the case where two cubies are placed correctly. If the the two correctly positioned edges are positioned directly across each other *algorithm*4*bA* is performed, this will result in two correctly positioned edge cubies next to each other.

The program checks which two edge cubies are positioned correctly and performs *algorithm*4*bA* accordingly. The edges are now positioned and orientated correctly and the program moves on to the next step.

13.1.6 Step 5a – Positioning the Last Layer Corners

In this step the corners in the last layer will be positioned correctly.

The initial action performed is a check of the number of correctly positioned corner cubies. The possible cases in this step are:

1. None of the corner cubies are positioned correctly.
2. One corner cubie is positioned correctly.
3. All corner cubies are positioned correctly.

If all four corner cubies are positioned correctly the program moves on to the next step. If none of the corners are positioned correctly *algorithm*5*aA* is performed. This will position one of the corner cubies correctly. When one corner is positioned correctly the program performs *algorithm 5aA* based on what corner is positioned correctly. Now the program will perform a check again to see if all the corners are positioned correctly. If all the four corners are not positioned correctly *algorithm 5aA* will be performed again. When that is done all the four corner cubies are positioned correctly and we can move on to the last step.

13.1.7 Step 5b – Completing the Last Layer

This is the last step of the solving process and only the corner cubies in the last layer need to be orientated correctly. This is more generally described in subsection 9.1.5

The algorithm takes one corner cubie at a time and checks its orientation. If the corner is not orientated correctly *algorithm*5*bA* is applied, this algorithm rotates a corner. When the corner is correctly orientated the **U** face is twisted so a new corner is placed in this position and the same procedure is applied. After four **U** turns and appliance of *algorithm*5*bA* the cube is finally solved.

13.2 Twist Reduction

As mentioned before our implementation of the beginner's algorithm is not as efficient as it can be. We can however take some steps towards improving its efficiency. One of these steps is removing unnecessary moves and replacing a move sequence with a shorter move sequence, which gives the same result.

An unnecessary move sequence can be performing a twist and just after that twist perform its inverse move. e.g. performing **U U'** or **U' U**. According to the theory of groups in section 7.2 two moves, which are each other's inverse give the same result as not performing the two moves.

It was necessary to be able to shorten the twists generally, so any move sequence can not be replaced with a shorter one in our program. An example of replacing a move sequence with a shorter one generally is replacing three of the same type of move with a single inverse move of that type. e.g. **U U U** also noted as **U2 U** or **U U2** can be replaced with a single inverse move **U'**. This is also described in the group theory section 7.2.

13.3 Statistics for Beginner's Algorithm

In order to get the average length of a solution of beginner's algorithm we must determine how many moves of scrambling must be applied in order to find the highest average number of twists required to solve the Rubik's Cube. By computing the average for different length of scramble sequences the graph quickly becomes steady at around 40 scrambles with 10.000 tested Rubik's Cubes (see figure 13.2 and appendix D). To be sure that we get a good result we will use 50 scrambles in the next tests.

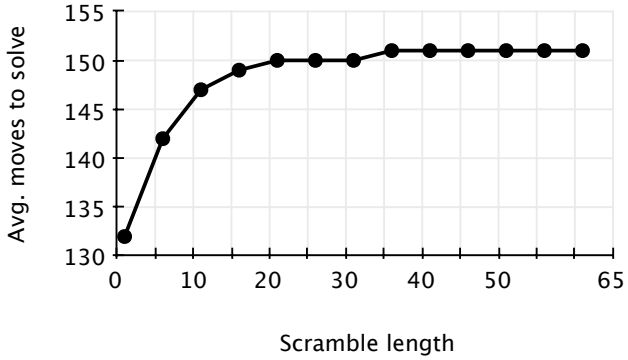


Figure 13.2: *The graph shows the amount of moves needed to solve a cube with x scrambles. The lines are only for visual effect and do not represent the average between the points.*

Now that the length of the needed scramble sequence is determined the average number of twists needed to solve the Rubik's Cube can be found. By scrambling and solving 10 millions Rubik's Cubes with our implementation of beginner's algorithm an average of 151 twists is obtained. A test using only 1 million Rubik's Cubes is run three times in order to ensure that the test is reliable. This test also showed an average of 151 moves in every run, meaning that this result is reproducible. The raw data from both tests can be found in appendix D.

The maximum length of the solution is 241 moves and the minimum length is 56 moves. The average time spent on each solve was:

$$\frac{502552 \text{ ms} + 490385 \text{ ms} + 498937 \text{ ms} + 4734126 \text{ ms}}{10^7 + 3 \cdot 10^6} = 0.478923077 \text{ ms}$$

The data was gathered on a computer operated by Windows 7 64 bits running on a 2.5 GHz AMD Quad Core processor (905e) with 4 GB DDR-3 RAM.

A peculiar note is that with a scramble of length one, the algorithm needs an average of 132 moves with a maximum of 155 and a minimum of 94 moves to solve it again. This illustrates the twist-wise inefficiency of this algorithm very well.

In this chapter the implementation of beginner's algorithm is described. Statistics about our implementation is gathered in order to compare this algorithm to our implementation of Kociemba's optimal solver.

14

Kociemba's Optimal Solver

Our implementation of Kociemba's optimal solver will be covered in this chapter. A general description of how we have chosen to implement Kociemba's optimal solver along with a detailed explanation of key points of the source code will be presented. Statistics about our implementation of Kociemba's optimal solver will be presented to make a comparison with beginner's algorithm in our discussion.

The basics of Kociemba's optimal solver have been covered in section 9.2. The algorithm which we are implementing differs from the original at one point: We are interested in the shortest move sequence to solve a scrambled cube, whereas the original Kociemba's optimal solver only finds the shortest length, not the actual sequence of moves [16].

Because of this we have chosen not to have a lookup table since it would have to contain 20 billion move sequences, which is impossible on today's computers. Considering that the lookup table, that only contains the length, would have a size of 4 GB [10]. A table containing the move sequences would have to be much larger.

An A move can be defined using 4 bits. The 4 bits gives $2^4 = 16$ different spaces and there are 10 A moves, which mean they will fit. To calculate the size of a lookup table containing the move sequence is just a matter of multiplying the size of one move with the number of moves in a sequence, e.g. a move sequence of the length 14 would have the size: $4 \cdot 14 = 56$ bits. This size is again multiplied with the number of positions which needs this length for solving. The size of a lookup table containing every move sequences to solve the Rubik's Cube inside **H** would be: 987 GB (see appendix B), which by far exceeds the average memory on today's computer [5] [2]. Instead of the lookup table our solver will have to try every move sequence inside **H** in order to find the shortest one that solves the Rubik's Cube.

A flowchart of our implementation is shown in figure 14.1. The solving method starts out with the Rubik's Cube in the position it has to be solved from. It begins by testing if the Rubik's Cube is solved, if this is the case the method will return. If the Rubik's Cube is not solved the method will test if the Rubik's Cube is in **H**, if this is the case it will call the method $solveFromH$. The method $solveFromH$ is built up using the same flow chart except the three boxes with a marked edge. The differences are described here.

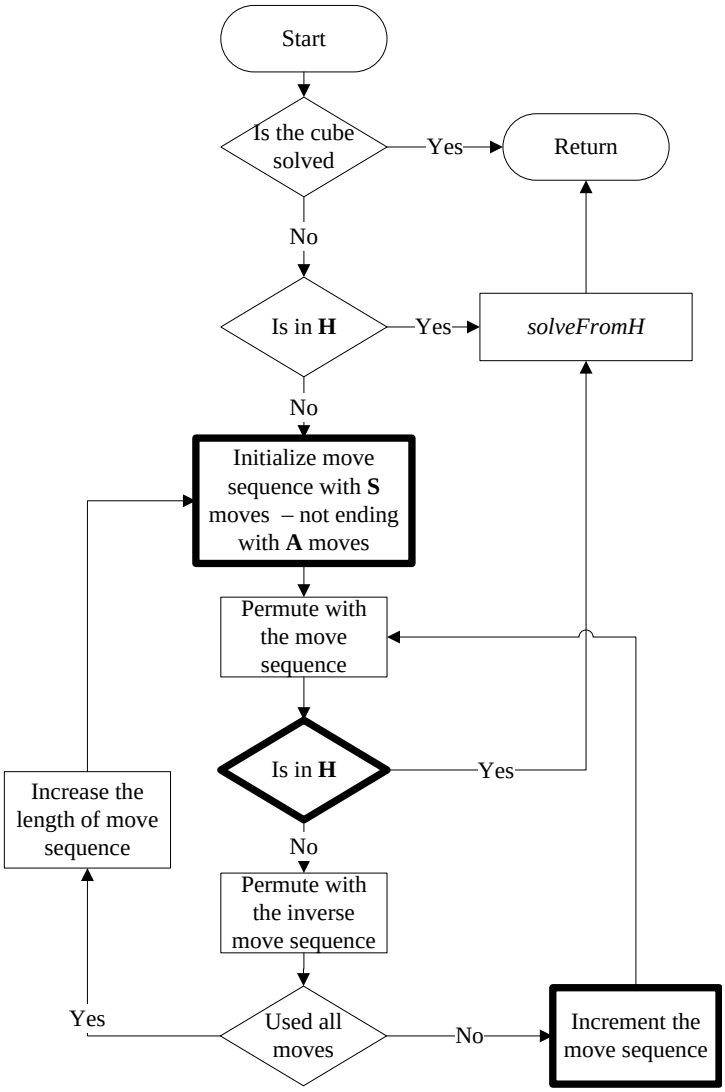


Figure 14.1: *A flow chart of our implementation of Kociemba's optimal solver.*

1. It creates a move sequence consisting of **A** moves instead of **S** moves.
2. It tests if the cube is solved instead of testing if it is in **H**.
3. It increments the move sequence with an **A** move (see next section for a description of move incremention).

If the Rubik's Cube is not in **H**, then a move sequence will be initialized with a predefined length of one in the first run. The move sequence will be made so two moves after each other are not on the same face (see section 7.2) and not ending with an **A** move. For instance an initialized move sequence of the length five would consist of the following moves: **U D U D F**.

Now the Rubik's Cube is permuted with the initialized move sequence. A new test is performed to check if Rubik's Cube is in **H**. If it is in **H** *solveFromH* will be called as it was before. If the Rubik's Cube is not in **H** the method will permute the inverse of the last move performed. If the last move cannot be incremented then it will try to increment the second last move ect. When a move has been incremented the Rubik's Cube will be permuted with this new move sequence. The test will then run again and a new move will be incremented untill the Rubik's Cube is in **H**. After all the moves in the move sequence have been incremented, then the method will increase the length of the move sequence and initialize a new move sequence. In the example from before the new move sequence would be: **U D U D U F**.

14.1 Incrementing a Move Sequence

Our implementation of Kociemba's optimal solver will be incrementing move sequences. This means that the move sequence will be changed to the next in a given set.

E.g. if a move sequence in **S** is **F R'**, then incrementing it will change the last move **R'** to the next **R2** – hence the new move sequence is **F R2**. This is very similar to incrementing an integer by hand; the last digit will become the next number in the decimal system until the last digit is 9. When this happens the second last digit is increased by one and the last digit is set to be the first number in the decimal system: 0.

So when our move sequence **F R2** is being incremented **R2** will become **U**, since **R2** is the last move in the set of moves **S**. **F** will then be increased to **F'**, this gives the new move sequence **F' U**.

14.2 The Solving Method

The method named *solve* (see code snippet 14.1 for a key point) acts like a main routine for the solving mechanism of Kociemba's optimal solver. By calling this method Kociemba's optimal solver will be applied to the Rubik's Cube.

The solver's approach to solving the Rubik's Cube is trying every move sequence not tried so far. If the cube is inside **H** it will start using only **A** moves. Every time a shorter move sequence is found it is saved.

The *solve* method starts with initializing five variables;

- The array *result* which will contain the final move sequence.

```
.
.
.
try {
  c = solveFromH(l - d);
  if (d + c.length < l) {
    curTime = System.currentTimeMillis();
    l = d + c.length;
    result = new MoveButtons[l];
    output.addTextln("The solutions of the length " + l + ". The
      solution is:");
    int j = 0;
    for ( ; j < d; j++) {
      result[j] = b[j];
      output.addText(b[j] + " ");
    }
    for (int k = 0 ; k < c.length; k++,j++) {
      result[j] = c[k];
      output.addText(c[k] + " ");
    }
    output.addTextln("");
    output.addTextln("Time spend: " + ((curTime - startTime)/1000)
      + " seconds");
  }
} catch (InvalidCubeException e) {}
...
```

Code snippet 14.1: *Key point in the solve method of Kociemba's optimal solver*

- The integer d which determines the search depth.
- The integer l which determines the length of the current solution.
- The array b which will contain the move sequence until the Rubik's Cube enters **H**. This is initialized with the length zero; an empty move sequence.
- The array c which will be the move sequence after the Rubik's Cube has entered **H**.

After the initialization the *solve* method calls the $solveFromH$, which solves the cube if the cube is in **H**. When the cube has been solved from **H** it tests if the length of the new solution is shorter than the old solution. This will be saved as the new solution. This can be seen in code snippet 14.1. If the cube is not in **H** the exception of the class $InvalidCubeException$ will be thrown. When this exception is caught it forces the *solve* method to skip to the next part of the algorithm.

After the code in the code snippet 14.1 has been run, the method will permute the Rubik's Cube back with the inverse of the last move. In first run this is redundant since the move sequence, b, is empty. When it has been permuted back the method $increaseWithSNotEndingWithA$ will be called. See section 14.3. The new increased move sequence will now be applied and the part displayed in code snippet 14.1 will be executed again.

When the method $increaseWithSNotEndingWithA$ has been through every move sequence of the length d, it will throw an exception of the type $UnableToIncreaseMoveSequenceException$. The *solve* method will catch this

exception and increment d. When d is incremented to l the method will return the array *result*, which contains the shortest move sequence to solve the given Rubik's Cube position.

14.3 The Move Incrementing Methods

The move incrementing methods are vital parts of Kociemba's optimal solver. They ensure that every possible move sequence is tested. The methods increment a move sequence. This means that it will find the next move in an enumset of all possible moves. See section 14.1.

When the method, named *increaseWithSNotEndingWithA*, is called it takes two parameters, a move sequence and an integer. The integer decides where in the move sequence the move should be incremented.

```
.
.
.
if (i == length - 1) {
  try {
    do {
      moveSequence[i] = (MoveButtons)S.toArray()[moveSequence[i].
        ordinal() + 1];
    } while(A.contains(moveSequence[i]));
    try {
      if(isSameFace(moveSequence[i], moveSequence[i-1])) {
        increaseWithSNotEndingWithA(moveSequence, i);
      } else {
        Cube.permute(cube, moveSequence[i]);
      }
    } catch (ArrayIndexOutOfBoundsException e2) {
      Cube.permute(cube, moveSequence[i]);
    }
    return;
  } catch (ArrayIndexOutOfBoundsException e) {
    moveSequence[i] = F;
    i--;
    try {
      Cube.permute(cube, moveSequence[i].invert());
      moveSequence[i].invert();
    } catch (ArrayIndexOutOfBoundsException e4) {
      throw new UnableToIncreaseMoveSequenceException();
    }
    increaseWithSNotEndingWithA(moveSequence, i);
    return;
  }
} else { ...
```

Code snippet 14.2: *Key point in the incrementing method of kociemba's optimal solver*

The last move performed on the cube can never be an **A** move since **H** is a closed group, which means that applying **A** moves to a cube in **H** will never make the cube leave **H** (see section 7.3). The part of the code that increments the last move is shown in code snippet 14.2. At first a **do-while** loop is executed. This loop adds a move from the enumset **S** (see code snippet 14.3, enumset **S**), which is not an **A** (see code snippet 14.3, enumset **A**) move.

After the loop has run a new method *isSameFace* is called. This method tests if the new move is on the same face as the move before. If the moves are on the same face the latter of the two moves will be incremented untill they are no longer on the same face. This test is only done on the last two moves, so they are not on the same face (see section 7.2).

When all the moves in the enumset **notA** (see code snippet 14.3, enumset **notA**) have been used it will start over from the beginning of the enumset **notA**.

```
private EnumSet<MoveButtons> S = EnumSet.of(U, UP ,U2, D, DP, D2, F
  , FP, F2, B, BP, B2, L, LP, L2, R, RP, R2);
private EnumSet<MoveButtons> A = EnumSet.of(U, UP ,U2, D, DP, D2,
  F2, B2, L2, R2);
private EnumSet<MoveButtons> notA = EnumSet.of(F, FP, B, BP, L, LP,
   R, RP);
```

Code snippet 14.3: *The definition of the enumsets S, A, and notA.*

All the code in code snippet 14.2 is executed if it is at the last move in the move sequence. If it is not at the last move the moves in the move sequences will be incremented with the moves from the enumset **S**. This ensures that all possible move sequences not ending with **A** will be tested.

There is a similar method named $increaseWithA$ that uses a very similar code. This is called from the $solveFromH$ method. The difference is that it only increments with moves from **A**.

14.4 Statistics for Kociamba's Optimal Solver

We will estimate how much time this solver will use to find an optimal solution to an average Rubik's Cube. The solver is an optimal solver and will always find an optimal solution (see section 9.2), which is why we will not gather statistics for its twist-wise efficiency.

In order to calculate the time used to solve a Rubik's Cube using Kociemba's optimal solver, we have gathered data from our application in form of time stamps whenever the solver has finished a search depth.

The data was gathered on a computer operated by Windows 7 64 bits running on a 2.5 GHz AMD Quad Core processor (905e) with 4 GB DDR-3 RAM. The data was gathered in two runs with these specifications and the raw data can be seen in appendix C.

14.4.1 First Phase

Two runs were made in the first phase – where the solver searches in **S**. The results are shown in table 14.1 and plotted in a coordinate system in figure 14.2 along with an approximation using an exponential function.

Based on figure 14.2 we assume that the time to finish a specific depth will evolve exponentially.

The figure only shows the last five data points since the first five are inaccurate due to small amount time. When the amount of computation time is small other processes can easily have interfered with the results.

Depth	Time to finish [ms]	
–	1st run	2nd run
0	0	0
1	0	16
2	0	31
3	47	78
4	281	296
5	1045	983
6	10920	10873
7	157451	158637
8	2358584	2361704
9	–	35447427

Table 14.1: *Data for computation time used to finish a depth in* $\boldsymbol{S}$ *moves of Kociemba's optimal solver, known as phase one*

As figure 14.2 shows the function for approximating the time needed to finish the search depth x is:

$$f(x) = 0.0018372421 \cdot 13.731997^x \text{ ms}$$

The R^2 value, 0.9884, is close to 1 and this tell us that the approximation is relatively good. The R^2 value is a way to express how well a set of data fits onto a specific approximation – the closer to 1, the better. We are only interested in a rough approximation of how long the solver needs to use to solve a Rubik's Cube, therefore our R^2 value is acceptable.

We are interested in approximating the time that it would take to solve a Rubik's Cube. Since the average Rubik's Cube needs 18 moves to be solved [11], we will estimate the time to compute this:

$$f(18) = 0.0018372421 \cdot 13.731997^{18} \text{ ms} \approx 5.5383 \cdot 10^{17} \text{ ms} \approx 17,600,000 \text{ years}$$

After this time our implementation will have solved an average scrambled Rubik's Cube with the optimal solution. However the solver will get into **H** a long time before this. As Michael Reid proved in 1995 [9] it takes a maximum of 12 twists for Kociemba's optimal solver to enter **H**, therefore we have chosen to look at the time it takes for the solver to search inside **H** in the second phase.

14.4.2 Second Phase

During the test our solver entered **H** many times, all the results are found in appendix C. We have chosen to look at the first five times the solver entered **H** and make an average from this data. The reason for the average is that the program, which we use to generate the tendency line, cannot generate an exponential tendency line from all the data points.

The five runs are shown in table 14.2.

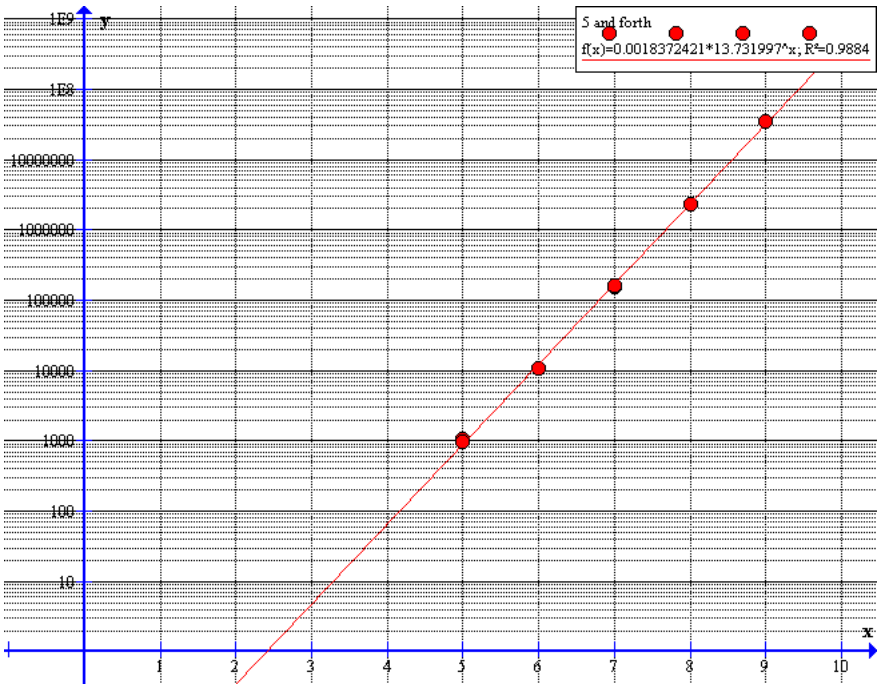


Figure 14.2: *The approximation of the time needed to finish the search depth in* ***S*** *moves. It is displayed on a logarithmic y-axis to show that an exponential function fits.*

As with the first phase we assume that an exponential function will make the best approximation to calculate the time spent to finish a given search depth.

Figure 14.3 shows that an exponential function fits quite well due to the relatively high R^2 value of 0.9098. The function to approximate the time needed to finish the search depth is seen to be:

$$f(x) = 0.003438036 \cdot 7.1431204^x \text{ ms}$$

Any Rubik's Cube that is inside **H** can be solved with 18 **A** moves. The time to finish a depth of 18 is approximately:

$$f(18) = 0.003438036 \cdot 7.1431204^{18} \text{ ms} \approx 8.0592 \cdot 10^{12} \text{ ms} \approx 256 \text{ years.}$$

This chapter has shown how we have implemented Kociemba's optimal solver with descriptions of key points of the source code. Statistics of our implementation of Kociemba's optimal solver is presented.

Depth	Time to finish [ms]					
–	1st time	2nd time	3rd time	4th time	5th time	Average
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	31	0	47	0	0	15.6
5	203	62	203	31	47	109.2
6	421	374	422	327	328	374.4
7	2153	2902	2153	2605	2605	2483.6
8	15787	22932	15866	20514	20420	19103.8
9	123506	181023	124083	161647	161336	150319
10	973348	1430164	977873	1276534	1275302	1186644.2
11	7687740	–	7894160	–	–	7790950
12	–	–	83980919	–	–	83980919

Table 14.2: *Data for computation time used to finish a depth in* ***A*** *moves of Kociemba's optimal solver in* ***A*** *moves, known as phase two*

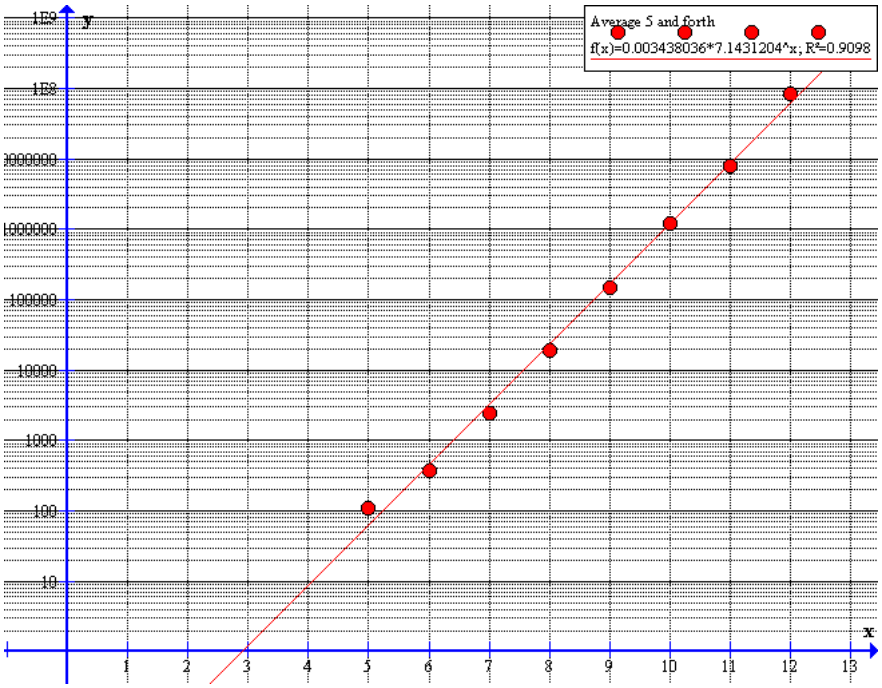


Figure 14.3: *The approximation of the time needed to finish the search depth in* ***A*** *moves. It is displayed on a logarithmic y-axis to show that an exponential function seem to fit.*

Part IV

Epilogue

15 Discussion

The two algorithms described in this report are very different. We will in this chapter describe those differences by using the gathered data from our implementations of them. The chosen programming language in the implementation and an alternative programming language will be discussed.

15.1 Comparing the Implemented Algorithms

The two algorithms implemented in our application differs a lot from each other when looking at efficiency. Our implementation of Kociemba's optimal solver is a twist-wise optimal solver, i.e. it uses the least amount of twists needed to solve any position (see section 9.2). The time-wise efficiency for Kociemba's optimal solver is not impressive. On a 2,5 GHz Quad Core, it would use approximately 18 million years to solve a Rubik's Cube 18 twists[1] away from the solved state (see section 14.4 for detailed calculations). On a test the Rubik's Cube will almost certainly have an optimal solution which goes into **H** and then starts to use **A** twists and thus lowering the search time substantially. Depth 18 in **H** is finished in approximately 256 years compared to the 18 million years it would take to solve the Rubik's Cube outside **H**, which is why the search time is lowered substantially.

Our implementation of beginner's algorithm is not designed for twist-wise or time-wise efficiency. However it uses less than a second to solve 1000 Rubik's Cubes (see section 13.3). Twist-wise it is way less efficient since it uses an average of 151 moves to solve a scrambled Rubik's Cube (see section 13.3).

Time-wise beginner's algorithm is more efficient than Kociemba's optimal solver. Specifically it is

$$\frac{5.5383 \cdot 10^{17} \text{ ms}}{0.478923077 \text{ ms}} \approx 1.2 \cdot 10^{18}$$

times faster on average. However this is based on that the shortest solution found by Kociemba's optimal solver is by not going into **H** and using **A** twists.

With respect to the number of twists Kociemba's optimal solver always gives the shortest twist sequence. Since it will simply brute force its way to a solution

[1] Most positions can be solved in 18 twists [11]

with a breadth first search. In our test of our implementation of beginner's algorithm we never got an optimal solution, which illustrates that beginner's algorithm will never be as twist-wise efficient as Kociemba's optimal solver.

From this discussion we can conclude that beginner's algorithm is a time-wise more efficient algorithm than Kociemba's optimal solver. This was expected since beginner's algorithm will use a "recipe" to solve the Rubik's Cube where Kociemba's optimal solver tests every possible twist sequence until it gets into **H**. This will in most cases require a lot of twists compared to the maximum of 241, which beginner's algorithm uses to solve a Rubik's Cube, which is scrambled with 50 twists. Furthermore we can conclude that beginner's algorithm is unable to solve a Rubik's Cube with an optimal solution. This is quite obvious since beginner's algorithm is divided into steps and it is very unlikely that these steps will go directly towards the solved state.

15.2 Another Programming Language

Java is an interpreted language [7], and therefore there is some advantages and disadvantages when using it.

One of the disadvantages with programs written in an interpreted language is that it executes slower compared to direct machine code executions [15]. Java is first compiled to virtual machine code (Java Bytecode) and then it is interpreted to machine code by a runtime application.

The advantage with using an interpreted programming language like Java is that it is very safe [7]. It can be executed on all platforms as long as the platform has a runtime application installed. E.g. if the program was written in C++ and compiled on one platform it would not be able to execute on a different platform [15].

If we focused our application to only work on one platform, e.g. Windows, we could write the application in C++. It has a faster execution time due to the fact that it compiles directly to machine code and this will reduce the solving time of our implemented algorithms [15].

16 Conclusion

Before we started the writing process we gathered information about what we found to be interesting aspects of the Rubik's Cube. These aspects included the different solving algorithms and some general mathematical theories that could be applied to the Rubik's Cube. After gathering sufficient information we formulated a problem statement. Our problem statement is as follows:

> *How have the upper and lower bounds of the Rubik's Cube progressed and how have they been proven?*
>
> *How efficient is Kociemba's optimal solver compared to beginner's algorithm and how can this be tested?*

The upper bound has been lowered several times, since Thistlewaite set the upper bound to 52 in 1981, while the lower bound has only been altered once since then (see chapter 5).

The upper bound of 22 has been found by dividing the positions of the Rubik's Cube into cosets. Rokicki's set solver (see chapter 10) was able to test if an entire set of positions needs 22 twists or less to solve the Rubik's Cube. The result showed that no Rubik's Cube position needs more than 22 twists to solve.

The lower bound was proven by testing all sequences of twists on the superflip position with a length shorter than 20, and finding that none of these solved this position. Thereby the lower bound was proven to be 20.

In order to compare the two implementations several tests must be run in a controlled environment with as little human interference as possible. Therefore an application, which is the only one to perform the solving algorithms is a good choice.

From our tests we can conclude that our implementation of beginner's algorithm is a time-wise more efficient algorithm than our implementation of Kociemba's optimal solver. Our implementation of beginner's algorithm is in average approximately $1.2 \cdot 10^{18}$ times faster than our implementation of Kociemba's optimal solver.

In average our implementation of beginner's algorithm uses 151 twists to solve a Rubik's Cube while our implementation of Kociamba's optimal solver always solves a Rubik's Cube with an optimal solution which is known to be

no more than 22. By this we conclude that our implementation of Kociemba's optimal solver is twist-wise more efficient than our implementation of beginner's algorithm.

17 Future Work

Our application is a proof of concept and can be improved in several ways. Here we will look at possible improvements to the two implemented algorithms.

17.1 Improving Beginner's Algorithm

The improvement of beginner's algorithm are described step by step.

A twist sequence which leads to the next step may be the most efficient for doing just that, but can potentially cause the entire twist sequence from the scrambled state to the solved state to be longer. i.e. a long partial twist sequence may lead to a shorter overall twist sequence. In order to utilize this a lot of analysis is necessary, which is nearly impossible for a person to perform and would take a large amount of time to implement. Additionally the algorithm still has to be beginner's algorithm, which means that the steps used in beginner's algorithm are also implemented and not just swapped for more efficient ones, which would defeat the purpose of implementing beginner's algorithm.

Beginner's algorithm is as mentioned before a very twist-wise inefficient solving algorithm. There are however quite a few areas in which the implementation can be improved. In the first step (see subsection 9.1.1) choosing the correct face to be the face of the first layer will decrease the number of twists needed to complete the first step, since a different part of the cross is assembled on the different faces.

In the second step (see subsection 9.1.2), where the corners needs to be positioned in the first layer, there are generally two ways to improve the algorithm. The first way to improve the twist sequence in this part is to choose the most efficient order of positioning the corners. Finding out which order is the most efficient one may seem to require a large amount of analysis, but since only four corners needs to be positioned a simulation can be used with advantage. There are generally $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ possible orders the corners can be positioned in, which makes the process of computing the shortest twist sequence a rather simple task. The other way to improve this step is to eliminate the repetition of an algorithm. The algorithm used for rotating a corner cubie in our implementation only rotates it in one direction. If an algorithm for rotating a corner cubie the other direction were implemented it would reduce the number of times the algorithms would be used.

In the third step (see subsection 9.1.3), where the edge cubies are positioned in the second layer, there is only one way to improve it; choosing the most efficient order to position the edge cubies. Since there are four edge cubies that need to be positioned, the logic used in improving the second step can be applied in the third step.

In the fourth step (see subsection 9.1.4), where the last layer cross is fully attained, there is only room for little improvement, if the algorithm should still be beginner's algorithm. As mentioned in (see subsection 9.1.4), there are three possible settings of the unsolved cross. In the position known as the opposite L, the reverse of the algorithm used in this step will skip the next setting and directly give the solved cross. This can potentially reduce the solving algorithm by six twists.

The fifth and final step of beginner's algorithm (see subsection 9.1.5) is divided into two methods in our implementation.

The first method of the final step, which positioned the remaining four corners correctly, only uses a single algorithm. This algorithm rotates three corner cubies' positions counterclockwise. Half the time the corner cubies' positions needs to be rotated clockwise, which means that this algorithm is used twice. Its reverse can be used to rotate the corner cubies' positions clockwise, which will in those cases when needed reduce the solving twist sequence by eight.

When using the half turn metric the second method of the final step cannot be improved while staying true to our description of beginner's algorithm (see section 9.1).

17.2 Improving Kociemba's Optimal Solver

Different improvements can be made to our implementation of Kociemba's optimal solver.

In the first phase of our implementation of Kociemba's optimal solver it checks if the Rubik's Cube is in **H** after every tested move sequence. This **H** is with respect to the primary faces (see subsection 9.2.2). It is also possible for the Rubik's Cube to be inside **H** with respect to the secondary or tertiary faces, see figure 17.1.

As with positions in primary **H**, the positions in secondary **H** must meet the following specifications: Every secondary facelet must be on the secondary faces, and the edges not in a secondary face must be orientated correctly. These specifications are met in figure 17.1.

When a shortest path to a position is found it could be saved in a lookup table. If Kociemba's optimal solver gets to a known position it can find the shortests path in the lookup table. If this is combined with the multiple **H** improvement, it could make the search time to get into **H** shorter.

The program is built up around one Rubik's Cube object. As a result of this the computer is only able to use one CPU core to work on the Rubik's Cube. It would be possible to make the application multithreaded in several ways. With some modifications it could be possible to make a copy of this Rubik's Cube object. This would make it possible to work on more than one Rubik's Cube object with the same starting position at a time.

When the application is working outside **H** it is working with **S** moves. Here it is possible to make a new thread for every depth the method searches in. In

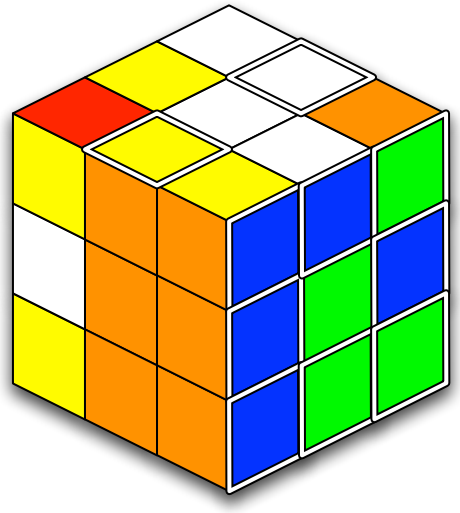


Figure 17.1: *A scrambled Rubik's Cube which is inside secondary* ***H****, but not in primary* ***H****.*

the same way it is possible to make a new thread for every search depth inside **H**.

When searching in general the moves in **S** and **A** could be divided into groups and a new thread could be made for these new groups of moves.

Part V

Appendices

A

E-mail Correspondence with H. Kociemba

```
From:   Herbert Kociemba [kociemba@t-online.de]
Sent:   16. marts 2010 16:44
To:     Alex Bondo Andersen
Subject:        Re: Rubik's Cube study at Aalborg University

Hello,

it is quite unusual to give away private informations to some "
    strangers", but ok,
here is some information.
I studied mathematics and physics (for "Lehramt an Gymnasien") at
    the Technische
Universit\"{a}t Darmstadt http://www.tu-darmstadt.de/ from
1974-1979 and am teacher for mathematics and physics since then at
    a Gymnasium. I
still live in Darmstadt. I am interested in Rubik's Cube since the
    beginning in
1980 - the same time were personal computers came up - and was
    immediately
interested to solve the cube algorithmically.
But it was not before 1990 that the PC- power was big enough to
    develop the ideas
and implementation for the two-phase-algorithm. I used an Atari St
    with 1 MB of
main memory for the first implementation and already got average
    solutions lengths
of about 21 moves....

If you have some other specific question, let me know.

Best regards

Herbert Kociemba
>
> Good day Herbert Kociemba,
>
> My name is Alex Bondo Andersen, I am attending Aalborg University
     in
> Denmark. My university group and I are working on a paper about
    the
> Rubik's Cube and are interested in using your webpage:
> http://kociemba.org/cube.htm as reference for a solving algorithm
    .
```

```
>
> If you are okay with us using your webpage as reference we would
    like
> to know a little about you in order to verify you as a credible
> source. So if you have the time it would be appreciated if you
    wrote
> which schools you have attended, where you live, which jobs you
    have
> had and why you are interested in the Rubik's Cube.
>
> In advance I would like to thank you for your time.
>
> Best Regards
>
> Group A215, Alex Bondo Andersen
>
```

B

Size of Lookup Table Containing Move Sequences

The size which a lookup table containing move sequences to solving any Rubik's Cube in **H** is found by adding the size of every move sequence which is used to solve a Rubik's Cube inside **H**. Since there are 10 move different moves in **A**, 4 bits will sufice for a single move. By taking the number of positions with a specific distances to e and multiplying this number by the distance and 4 the total number of bits needed for saving every position of this distance is found. The size of the entire lookup table will then be [16]:
$moves[0] := 1$
$moves[1] := 10$
$moves[2] := 67$
$moves[3] := 456$
$moves[4] := 3079$
$moves[5] := 19948$
$moves[6] := 123074$
$moves[7] := 736850$
$moves[8] := 4185118$
$moves[9] := 22630733$
$moves[10] := 116767872$
$moves[11] := 552538680$
$moves[12] := 2176344160$
$moves[13] := 5627785188$
$moves[14] := 7172925794$
$moves[15] := 3608731814$
$moves[16] := 224058996$
$moves[17] := 1575608$
$moves[18] := 1352$

$$size := \sum_{i=0}^{18} 4 \cdot i \cdot moves[i] = 1059719204004$$

$$\frac{size}{2^{10^3}} = 986.9404174$$

C

Test Results for Kociemba's Optimal Solver

```
Behold the Cube
Scrambling:  B B' B' F2 D2 D2 L' L' B' L R' B L2 D2 L2 B U2 R' R2
    B2 B2 D U D L2 B2 U' U2 R' F2 R B' D2 U2 D2 U2 R' F' U2 F U B2
    R' F B D' U D2 U U2
Solving with Kociemba's improved algorithm, please wait.
Try solving with depth: 0. Time spend: 0 milliseconds
Try solving with depth: 1. Time spend: 0 milliseconds
Try solving with depth: 2. Time spend: 0 milliseconds
Try solving with depth: 3. Time spend: 0 milliseconds
Try solving with depth: 4. Time spend: 47 milliseconds
Try solving with depth: 5. Time spend: 281 milliseconds
Try solving with depth: 6. Time spend: 1045 milliseconds
Try solving with depth: 7. Time spend: 10920 milliseconds
Try solving with depth: 8. Time spend: 157451 milliseconds
Try solving with depth: 9. Time spend: 2358584 milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 0 milliseconds
Solving in H, with depth: 5. Time spend inside H: 31 milliseconds
Solving in H, with depth: 6. Time spend inside H: 203 milliseconds
Solving in H, with depth: 7. Time spend inside H: 421 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2153 milliseconds
Solving in H, with depth: 9. Time spend inside H: 15787
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 123506
    milliseconds
Solving in H, with depth: 11. Time spend inside H: 973348
    milliseconds
Solving in H, with depth: 12. Time spend inside H: 7687740
    milliseconds
The solutions of the length 21. The solution is:
U  R2  D'  L'  F'  L'  D2  B'  L  B2  D2  R2  U'  R2  B2  U  F2  R2
     U  L2  D
Time spend: 55159 seconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 0 milliseconds
Solving in H, with depth: 5. Time spend inside H: 0 milliseconds
Solving in H, with depth: 6. Time spend inside H: 62 milliseconds
Solving in H, with depth: 7. Time spend inside H: 374 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2902 milliseconds
```

```
Solving in H, with depth: 9. Time spend inside H: 22932
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 181023
    milliseconds
Solving in H, with depth: 11. Time spend inside H: 1430164
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 0 milliseconds
Solving in H, with depth: 5. Time spend inside H: 0 milliseconds
Solving in H, with depth: 6. Time spend inside H: 31 milliseconds
Solving in H, with depth: 7. Time spend inside H: 327 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2605 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20514
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 161647
    milliseconds
Solving in H, with depth: 11. Time spend inside H: 1276534
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 0 milliseconds
Solving in H, with depth: 5. Time spend inside H: 0 milliseconds
Solving in H, with depth: 6. Time spend inside H: 47 milliseconds
Solving in H, with depth: 7. Time spend inside H: 328 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2605 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20420
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 161336
    milliseconds
Solving in H, with depth: 11. Time spend inside H: 1275302
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 0 milliseconds
Solving in H, with depth: 5. Time spend inside H: 0 milliseconds
Solving in H, with depth: 6. Time spend inside H: 47 milliseconds
Solving in H, with depth: 7. Time spend inside H: 328 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2590 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20530
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 162162
    milliseconds
Solving in H, with depth: 11. Time spend inside H: 1280793
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 0 milliseconds
Solving in H, with depth: 5. Time spend inside H: 0 milliseconds
Solving in H, with depth: 6. Time spend inside H: 47 milliseconds
Solving in H, with depth: 7. Time spend inside H: 328 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2605 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20592
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 162740
    milliseconds
Solving in H, with depth: 11. Time spend inside H: 1284881
    milliseconds
```

```
Try solving with depth: 10. Time spend: 146572164 milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 1 milliseconds
Solving in H, with depth: 4. Time spend inside H: 2 milliseconds
Solving in H, with depth: 5. Time spend inside H: 6 milliseconds
Solving in H, with depth: 6. Time spend inside H: 44 milliseconds
Solving in H, with depth: 7. Time spend inside H: 335 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2644 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20844
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 163944
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 1 milliseconds
Solving in H, with depth: 5. Time spend inside H: 6 milliseconds
Solving in H, with depth: 6. Time spend inside H: 43 milliseconds
Solving in H, with depth: 7. Time spend inside H: 335 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2632 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20781
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 164164
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 1 milliseconds
Solving in H, with depth: 3. Time spend inside H: 1 milliseconds
Solving in H, with depth: 4. Time spend inside H: 2 milliseconds
Solving in H, with depth: 5. Time spend inside H: 7 milliseconds
Solving in H, with depth: 6. Time spend inside H: 45 milliseconds
Solving in H, with depth: 7. Time spend inside H: 335 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2630 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20733
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 163719
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 1 milliseconds
Solving in H, with depth: 3. Time spend inside H: 1 milliseconds
Solving in H, with depth: 4. Time spend inside H: 2 milliseconds
Solving in H, with depth: 5. Time spend inside H: 7 milliseconds
Solving in H, with depth: 6. Time spend inside H: 44 milliseconds
Solving in H, with depth: 7. Time spend inside H: 336 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2634 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20765
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 163860
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 1 milliseconds
Solving in H, with depth: 3. Time spend inside H: 1 milliseconds
Solving in H, with depth: 4. Time spend inside H: 2 milliseconds
Solving in H, with depth: 5. Time spend inside H: 7 milliseconds
Solving in H, with depth: 6. Time spend inside H: 45 milliseconds
Solving in H, with depth: 7. Time spend inside H: 335 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2623 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20724
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 163708
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
```

```
Solving in H, with depth: 2. Time spend inside H: 1 milliseconds
Solving in H, with depth: 3. Time spend inside H: 1 milliseconds
Solving in H, with depth: 4. Time spend inside H: 2 milliseconds
Solving in H, with depth: 5. Time spend inside H: 6 milliseconds
Solving in H, with depth: 6. Time spend inside H: 43 milliseconds
Solving in H, with depth: 7. Time spend inside H: 333 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2628 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20758
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 163886
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 1 milliseconds
Solving in H, with depth: 3. Time spend inside H: 1 milliseconds
Solving in H, with depth: 4. Time spend inside H: 2 milliseconds
Solving in H, with depth: 5. Time spend inside H: 8 milliseconds
Solving in H, with depth: 6. Time spend inside H: 47 milliseconds
Solving in H, with depth: 7. Time spend inside H: 338 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2635 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20748
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 163702
    milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 1 milliseconds
Solving in H, with depth: 3. Time spend inside H: 1 milliseconds
Solving in H, with depth: 4. Time spend inside H: 2 milliseconds
Solving in H, with depth: 5. Time spend inside H: 7 milliseconds
Solving in H, with depth: 6. Time spend inside H: 44 milliseconds
Solving in H, with depth: 7. Time spend inside H: 336 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2633 milliseconds
Solving in H, with depth: 9. Time spend inside H: 20747
    milliseconds
Solving in H, with depth: 10. Time spend inside H: 163904
    milliseconds

Behold the Cube
Scrambling:  D' U' L2 U2 R R2 R' L' R2 L U' D2 R2 D' F' U' R B D' B
     F B2 U2 L2 D L' D D' R B B R2 R2 U2 D R2 L' F2 D' U' F2 D2 B R
    ' R2 U2 F F B B2
Solving with Kociemba's improved algorithm, please wait.
Try solving with depth: 0. Time spend: 0 milliseconds
Try solving with depth: 1. Time spend: 0 milliseconds
Try solving with depth: 2. Time spend: 16 milliseconds
Try solving with depth: 3. Time spend: 31 milliseconds
Try solving with depth: 4. Time spend: 78 milliseconds
Try solving with depth: 5. Time spend: 296 milliseconds
Try solving with depth: 6. Time spend: 983 milliseconds
Try solving with depth: 7. Time spend: 10873 milliseconds
Try solving with depth: 8. Time spend: 158637 milliseconds
Try solving with depth: 9. Time spend: 2361704 milliseconds
Try solving with depth: 10. Time spend: 35447427 milliseconds
Solving in H, with depth: 1. Time spend inside H: 0 milliseconds
Solving in H, with depth: 2. Time spend inside H: 0 milliseconds
Solving in H, with depth: 3. Time spend inside H: 0 milliseconds
Solving in H, with depth: 4. Time spend inside H: 0 milliseconds
Solving in H, with depth: 5. Time spend inside H: 47 milliseconds
Solving in H, with depth: 6. Time spend inside H: 203 milliseconds
Solving in H, with depth: 7. Time spend inside H: 422 milliseconds
Solving in H, with depth: 8. Time spend inside H: 2153 milliseconds
Solving in H, with depth: 9. Time spend inside H: 15866
    milliseconds
```

```
Solving in H, with depth: 10. Time spend inside H: 124083
    milliseconds
Solving in H, with depth: 11. Time spend inside H: 977873
    milliseconds
Solving in H, with depth: 12. Time spend inside H: 7894160
    milliseconds
Solving in H, with depth: 13. Time spend inside H: 83980919
    milliseconds
```

Test Results for Beginner's Algorithm

```
Calculating statistics for Beginners Algorithm:
Scrambles: 1 Runs: 10000 in average 132 moves.
Max: 155 min: 94 Time 7040
Scrambles: 6 Runs: 10000 in average 142 moves.
Max: 221 min: 0 Time 7164
Scrambles: 11 Runs: 10000 in average 147 moves.
Max: 218 min: 63 Time 7362
Scrambles: 16 Runs: 10000 in average 149 moves.
Max: 215 min: 73 Time 7440
Scrambles: 21 Runs: 10000 in average 150 moves.
Max: 214 min: 76 Time 7479
Scrambles: 26 Runs: 10000 in average 150 moves.
Max: 220 min: 80 Time 7489
Scrambles: 31 Runs: 10000 in average 150 moves.
Max: 220 min: 80 Time 7471
Scrambles: 36 Runs: 10000 in average 151 moves.
Max: 212 min: 64 Time 7503
Scrambles: 41 Runs: 10000 in average 151 moves.
Max: 213 min: 83 Time 7507
Scrambles: 46 Runs: 10000 in average 151 moves.
Max: 221 min: 83 Time 7516
Scrambles: 51 Runs: 10000 in average 151 moves.
Max: 222 min: 74 Time 7484
Scrambles: 56 Runs: 10000 in average 151 moves.
Max: 223 min: 83 Time 7524
Scrambles: 61 Runs: 10000 in average 151 moves.
Max: 214 min: 74 Time 7450
Scrambles: 66 Runs: 10000 in average 151 moves.
Max: 210 min: 71 Time 7533
Scrambles: 71 Runs: 10000 in average 151 moves.
Max: 218 min: 75 Time 7485
Scrambles: 76 Runs: 10000 in average 151 moves.
Max: 216 min: 82 Time 7526
Scrambles: 81 Runs: 10000 in average 150 moves.
Max: 221 min: 79 Time 7529
Scrambles: 86 Runs: 10000 in average 151 moves.
Max: 217 min: 78 Time 7480
Scrambles: 91 Runs: 10000 in average 151 moves.
Max: 217 min: 81 Time 7470
Scrambles: 96 Runs: 10000 in average 151 moves.
Max: 217 min: 78 Time 7495
Scrambles: 101 Runs: 10000 in average 151 moves.
Max: 217 min: 82 Time 7494
```

```
Behold the Cube
Calculating statistics for Beginners Algorithm:
Scrambles: 50 Runs: 1000000 in average 151 moves.
Max: 230 min: 59 Time 502552

Behold the Cube
Calculating statistics for Beginners Algorithm:
Scrambles: 50 Runs: 1000000 in average 151 moves.
Max: 230 min: 65 Time 490385

Behold the Cube
Calculating statistics for Beginners Algorithm:
Scrambles: 50 Runs: 1000000 in average 151 moves.
Max: 232 min: 56 Time 498937

Behold the Cube
Calculating statistics for Beginners Algorithm:
Scrambles: 50 Runs: 10000000 in average 151 moves.
Max: 241 min: 58 Time 4734126
```

Bibliography

[1] Hardeep Aiden. Anything but square: from magic squares to sudoku. WWW, March 2006. URL http://plus.maths.org/issue38/features/aiden/. Last viewed: 16/2.

[2] Aller Media A/S. Edb priser. WWW, April 2010. URL http://www.edbpriser.dk. Last viewed: 26/4.

[3] Encyclopædia Britannica. Encyclopædia britannica online. WWW, 2010. URL http://www.britannica.com/EBchecked/topic/369194/mathematics. Last viewed: 5/4.

[4] Alan Chang. Learn2cube. WWW, February 2009. URL http://www.learn2cube.com. Last viewed: 18/3.

[5] Ashlyn Crowsey. How much memory does an average computer have? WWW, October 2009. URL http://www.answerbag.com/q_view/1792480. Last viewed: 26/4.

[6] Technische Universität Darmstadt. Technische universität darmstadt. WWW, April 2010. URL http://www.tu-darmstadt.de/. Last viewed: 13/4.

[7] David J. Eck. Javanotes 5.1.1. WWW, December 2009. URL http://math.hws.edu/javanotes/. Last viewed: 20/5.

[8] David Joyner. *Adventures in group theory: Rubik's Cube, Merlin's machine, and other mathematical toys.* The Johns Hopkins University Press, 2002. ISB: 0-8018-6945-5.

[9] Knowledgerush. Optimal solutions for rubik's cube. WWW, 2009. URL http://www.knowledgerush.com/kr/encyclopedia/Optimal_solutions_for_Rubik%27s_Cube/. Last viewed: 17/5.

[10] Herbert Kociemba. *Cube Explorer 3*, 2005.

[11] Herbert Kociemba. Cube explorer 4.64. WWW, 2009. URL http://kociemba.org/cube.htm.

[12] Mogens Esrom Larsen. *Rubiks terning*. Nyt Nordisk Forlag Arnold Busck, 1981.

[13] Jelsoft Enterprises Ltd. Speedsolving the rubik's cube & other puzzles. WWW, 2010. URL http://www.speedsolving.com/forum/index.php. Last viewed: 18/3.

[14] Jelsoft Enterprises Ltd. Superflip - speedsolving.com wiki. WWW, January 2010. URL http://www.speedsolving.com/wiki/index.php/Superflip. Last viewed: 23/3.

[15] John S. Riley. Interpreted vs. compiled languages. WWW, 2005. URL http://www.dsbscience.com/freepubs/start_programming/node6.html. Last viewed: 20/5.

[16] Tomas Rocicki. Twenty-two moves suffice for rubik's cube. *Mathematical Entertainments*, November 2009.

[17] Tomas Rokicki. Computers and the cube. WWW, 2009. URL cube.stanford.edu/class/files/rokicki_cubecomp.pdf. Last viewed: 18/5.

[18] Kenneth H. Rosen. *Discrete Mathematics And Its Applications*. McGraw-Hill, sixth edition, 2007. ISBN-13: 978-007-124474-9.

[19] Ernö Rubik, Tamás Varga, Cerzson Kéri, György Marx, and Tamás Vekerdy. *Rubik's Cubic Compendium*. Oxford University Press, 1987. ISBN: 0-19-853202-4.

[20] Jaap Scherphuis. Jaap's puzzle page. WWW, 2010. URL http://www.jaapsch.net/puzzles/. Last viewed: 16/2.

[21] Jaap Scherphuis. Jaap's puzzle page. WWW, 2010. URL http://www.jaapsch.net/puzzles/thistle.htm. Last viewed: 6/5.

[22] David Singmaster. The unreasonable utility of recreational mathematics. WWW, December 1998. URL http://www.eldar.org/~problemi/singmast/ecmutil.html. Last viewed: 15/2.

[23] Juan Soulie. Boolean operations. WWW, October 2009. URL http://www.cplusplus.com/doc/boolean/. Last viewed: 19/4.

[24] Dik t. Winter. Kociembas algorithm. Emails, December 1992. URL http://www.math.ucf.edu/~reid/Rubik/Cubelovers/cube-mail-8. Last viewed: 15/4.

[25] Charles W. Trigg. What is recreational mathematics? *Mathematics Magazine*, 51:18 – 21, Januar 1978. URL http://www.jstor.org/stable/2689642?seq=1&cookieSet=1.

[26] Ron van Bruchem, Tyson Mao, and Masayuki Akimoto. World cube association. WWW, February 2010. URL http://www.worldcubeassociation.org/. Last viewed: 16/2.