BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE IN ENGLISH

DIPLOMA THESIS

Solving Techniques for Rubik's Cube in Software and Robotics Environment

Supervisor
Lect. Nechita Mihai-Simion

Author
Măierean Mircea

2026

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENŢĂ

Tehnici de rezolvare a cubului Rubik în software și în robotică

Conducător științific
Lect. Nechita Mihai-Simion

Absolvent
Măierean Mircea

2026

ABSTRACT

The Rubik's Cube, the world's bestselling 3D puzzle, was invented in 1974 by Ernő Rubik and has since become an enduring icon of popular culture. Despite its widespread popularity, many individuals remain unable to solve the cube due to the complexity of its mechanics and the algorithms required. This thesis addresses this challenge by presenting a complete technical solution designed to assist users in solving the Rubik's Cube efficiently.

The thesis is structured into two main components: software and hardware. On the software side, an interface integrates image analysis of a scrambled cube, identify its current state, and computes the optimal sequence of moves to achieve the solved configuration. A significant contribution is the creation of a web application that enables users to interactively scan their own cubes, receive precise move-by-move instructions, and visualize the solution process.

The hardware component showcases a robotic system capable of physically manipulating a real Rubik's Cube. This robot autonomously performs the computed moves, demonstrating the feasibility of a fully automated solving system. The combined software-hardware approach highlights the originality of the work, integrating communication between services, algorithmic optimization, and mechanical actuation to bridge the gap between virtual solutions and real-world execution.

Overall, the thesis not only contributes a novel tool to assist and educate users but also explores the interdisciplinary challenges of combining software algorithms with physical hardware. By offering both an accessible user interface and a functioning robotic prototype, this work demonstrates practical innovation with potential applications in education, entertainment, and automation.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

The Rubik's Cube, the world's bestselling puzzle, is a 3D mechanical brainteaser invented in 1974 by Ernő Rubik, a Hungarian sculptor and professor of architecture. After its international release in 1980, the cube became a global sensation, securing its place as a lasting icon of problem-solving and intellectual challenge.

Despite its straightforward appearance — a cube with six coloured faces — the Rubik's Cube conceals immense complexity for beginners. With over 43 quintillion possible permutations, it poses a combinatorial challenge that many find daunting. Over the decades, the cube has inspired a passionate worldwide community, from casual hobbyists to elite speedcubers breaking world records.

Yet for the majority of people, the Rubik's Cube remains an unsolved mystery. Many individuals have experimented with the cube only to abandon it in frustration, overwhelmed by its nonintuitive moves, complex patterns, and the need for algorithmic thinking. While extensive resources such as books, tutorials, and online videos exist, they often assume prior knowledge, persistence, or a strong technical inclination — making them inaccessible to the average beginner.

This widespread disconnect between the cube's popularity and the general public's ability to solve it highlights an important motivational context for this work. It raises the question: how can we make the Rubik's Cube more approachable and its solving process more transparent for nonexperts? Moreover, it invites exploration into how technology can bridge this gap, creating learning environments where even complex puzzles become engaging and accessible.

This thesis is situated within this context, driven by the cultural and intellectual significance of the Rubik's Cube and the motivation to lower the barriers to entry for those eager to engage with it. By examining both the software and hardware dimensions of interactive puzzle solving, the work builds on decades of fascination

with the cube, aiming to explore innovative ways to combine traditional challenges with modern technological aids.

1.2 Objectives

The main objectives of this thesis are:

- To develop a software system capable of analysing images of a scrambled Rubik's Cube and accurately detecting its current state.
- To determine an optimal sequence of moves needed to solve the cube from any given configuration.
- To design a user-friendly web application that allows users to scan their own cubes and receive clear, step-by-step solving instructions.
- To construct a robotic system capable of physically manipulating a real Rubik's Cube, autonomously performing the computed solving steps.
- To integrate the software and hardware components into a cohesive system that demonstrates both virtual and physical solving capabilities.
- To evaluate the accuracy, efficiency, and user experience of the developed system through testing and practical demonstrations.
- To contribute original insights into the combination of computer vision, algorithmic puzzle solving, and robotics for interactive educational tools.

1.3 Thesis Structure

The thesis is organized into several chapters, each focusing on a distinct component of the research and development process.

The first chapter provides the context, motivation, and objectives behind this work, offering an overview of the Rubik's Cube's cultural significance and the challenges it presents to both casual users and puzzle enthusiasts.

The next chapter presents a review of the literature and related work, summarizing existing research in the fields of puzzle solving, algorithmic approaches, and robotics. This establishes the background knowledge required for the development of this project and highlights relevant prior contributions that shaped the direction of this thesis.

Following these introductory sections, the thesis explores the theoretical and practical aspects of solving the Rubik's Cube, detailing the established methods and

step-by-step strategies, including notation systems and solution algorithms. This lays the groundwork for understanding the technical implementation of the system.

Subsequent chapters focus on the development of the system itself. The component detection chapter explains how colour detection and image analysis are employed to interpret the state of the cube. The practical usage chapter then discusses the design and implementation of the web application, describing its requirements, specifications, and user interface.

The hardware aspects are covered in the robot chapter, which describes the hardware components, such as the Raspberry Pi, stepper motors, and motor drivers, as well as the 3D modelling work necessary for constructing the robot.

The performance measurement chapter presents the experimental evaluation of the system, detailing how its accuracy, efficiency, and practical usability were assessed and what insights were gained from the tests conducted.

Finally, the conclusions chapter summarizes the key findings, contributions, and outcomes of the thesis. It also reflects on the project's limitations and outlines potential directions for future work and improvements.

This structured approach ensures that the thesis offers a complete exploration of the topic, covering theoretical background, practical implementation, experimental evaluation, and forward-looking perspectives, thereby providing a comprehensive understanding of the work undertaken.

During the preparation of this work the author used ChatGPT in order to ensure awkward phrasing throughout this thesis is avoided. After using this tool/service, the author reviewed and edited the content as needed and takes full responsibility for the content of the thesis.

Chapter 2

Literature Review and Related Work

Since its invention, the Rubik's Cube has captivated a wide range of people. Some were intrigued by the physical challenge of solving it by hand, navigating its standard set of movements, while others approached it from a computational perspective, aiming to discover faster and more efficient solving methods. Its widespread popularity has drawn the attention of developers and researchers alike—some created applications that solve the cube based on a user-input configuration, while others explored more advanced territory, developing libraries and algorithms capable of handling complex cube-solving logic.

2.1 Singmaster Notation

Most of the available solutions for solving a cube use a notation developed by David Singmaster that denotes a sequence of moves. The nature of this notation is not fixed, as it can be relatively applied by the solver, no matter how an individual holds a cube nor the colours the puzzle has. For each move, a letter is associated, and describe a *clockwise rotation* of the respective face. Each of these letters can be followed by a prime symbol (′), indicating that the move has to be done *anticlockwise*. A letter that is followed by a 2 indicates the rotation has to be done twice.

The main notations used to represent face rotations of the Rubik's Cube are shown below. Each description is followed by a visual illustration that can be referenced using the corresponding figure number.

F (Front): the side currently facing the solver, as shown in Fig. 2.1.

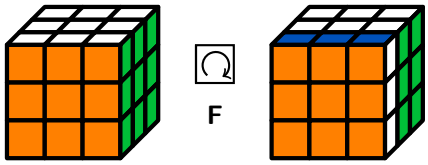

Figure 2.1: Rotation of the Front face (F)

B (Back): the side opposite the front, as shown in Fig. 2.2.

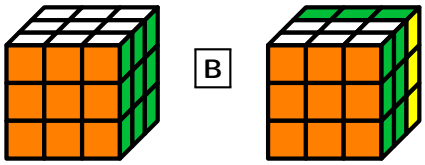

Figure 2.2: Rotation of the Back face (B)

U (Up): the side on top of the cube, as shown in Fig. 2.3.

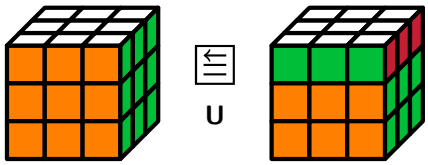

Figure 2.3: Rotation of the Up face (U)

D (Down): the side underneath the cube, as shown in Fig. 2.4.

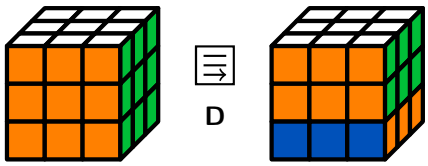

Figure 2.4: Rotation of the Down face (D)

L (Left): the side directly to the left of the front, as shown in Fig. 2.5.

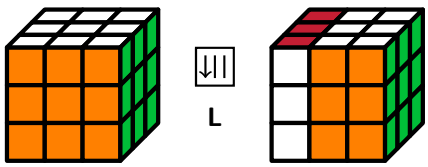

Figure 2.5: Rotation of the Left face (L)

R (Right): the side directly to the right of the front, as shown in Fig. 2.6.

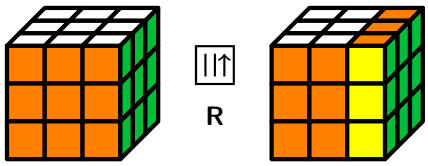

Figure 2.6: Rotation of the Right face (R)

M (Middle): the middle of the cube is moved shown in Fig. 2.7.

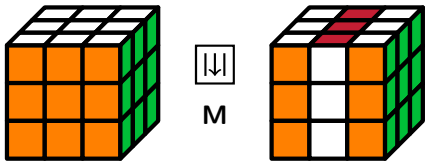

Figure 2.7: Middle Slice (M)

X: whole cube is rotated with respect to the X axis moved shown in Fig. 2.8.

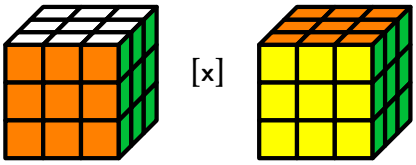

Figure 2.8: X Rotation

Y: whole cube is rotated with respect to the Y axis moved shown in Fig. 2.9.

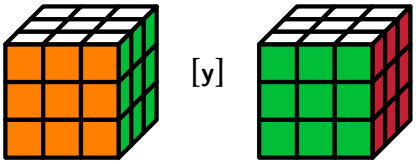

Figure 2.9: Y Rotation

Z: whole cube is rotated with respect to the Z axis moved shown in Fig. 2.10.

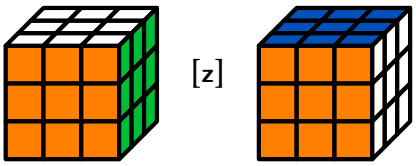

Figure 2.10: Z Rotation

In Figure Fig. 2.11, a sequence of moves is described by **U' R' F** applied on a solved cube, with the orange face in front and green on the right should bring the cube in the following configuration:

From now on, a continuous sequence of moves will be called an *algorithm*.

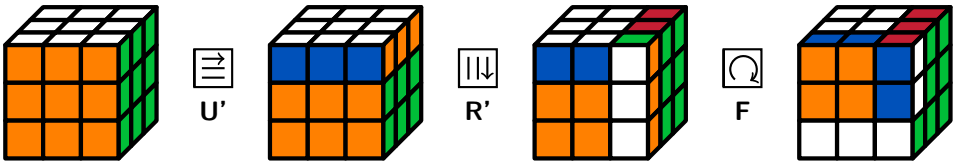

Figure 2.11: The rotations **U'**, **R'**, **F** on a solved cube.

2.2 Human Solutions

The Rubik's Cube was designed with a solution accessible to humans, allowing virtually anyone to solve it with some practice and determination. While the original solving methods were standardized and even patented, the rise of the speedcubing community has led to the development of much faster and more efficient techniques. These advanced methods are significantly more complex and have enabled solvers to achieve astonishing solve times. As of now, the fastest officially recorded solve is an incredible 3.05 seconds, set by Xuanyi Geng, a 7-year-old from China.

2.2.1 Layer by Layer

The Layer-by-Layer method is arguably the most popular and widely used approach for solving the Rubik's Cube. It breaks the solution into seven sequential steps, each relying on specific algorithms to solve different sections of the cube in order. This method is simple enough for beginners to grasp while remaining structurally sound for programmatic implementation. More details about each step are provided in Chapter 3, as this technique sets the foundation for the system's ability to solve a scrambled cube.

2.2.2 CFOP

The CFOP Method (Cross, F2L, OLL, PLL) is one of the most efficient and widely adopted solving techniques present in the speedcubing community. It is based on building by layers, an improvement of the Layer by Layer method. It introduces more advanced algorithms and pair-solving strategies. Like its predecessor, it starts with the same first step, forming a cross on one of the faces. Next, it is followed by solving the First Two Layers (F2L), by inserting corner-edge pairs for the face that has the cross positioned on it. Following this, the next step is the Orientation of the Last Layer (OLL), aligning all the top face colours. Finally, Permutation of the Last Layer (PLL) positions all the remaining pieces of the last face to complete the cube. [Rom17a]

Table 2.1: Number of Cases for Each Step in the CFOP Method

Step	Name	Number of Cases
1	Cross (White Cross)	Intuitive (no fixed cases)
2	F2L (First Two Layers)	41
3	OLL (Orientation of Last Layer)	57
4	PLL (Permutation of Last Layer)	21
Total		**119 algorithms**

Although CFOP requires memorizing a large set of algorithms—especially for OLL and PLL—it offers high efficiency and low move counts, making it the preferred choice for most world-class speedcubers.

2.2.3 The Petrus Method

The **Petrus Method** is a block-building approach to solving the Rubik's Cube, developed by Lars Petrus in the early 1980s. Unlike layer-based methods such as CFOP, Petrus emphasizes solving flexibility, early edge orientation, and low move count. The method consists of seven steps that encourage intuitive solving and spatial reasoning.

Steps of the Petrus Method

1. Build a 2×2×2 block.
2. Expand the block into a 2×2×3.
3. Orient all edges (ensures freedom in further moves).
4. Complete the First Two Layers (F2L).
5. Solve the last layer corners.
6. Permute corners.
7. Permute edges.

A notable advantage of the Petrus method is the early orientation of edges in Step 3, which allows full face turns during the rest of the solve without disrupting orientation. The initial block-building approach also avoids many of the move restrictions encountered in layer-based methods.

Despite being less popular in competitive speedcubing, the Petrus method remains a valuable solving technique for understanding cube theory and optimizing move efficiency. [Spe24a]

2.2.4 ZZ Method

The **ZZ Method** is an advanced and rotationless approach to solving the Rubik's Cube, created by Zbigniew Zborowski. It combines edge orientation and block building to enable efficient and ergonomic solves, often using only R, U, and L moves after the initial step. The method begins with a unique phase called EO-Line, where all edges are oriented and two cross edges are placed simultaneously. This allows for smooth First Two Layer (F2L) solving with minimal regrips. The ZZ method emphasizes lookahead and turning efficiency, making it popular among solvers who value fluid solves and experimental techniques. [Spe24c]

Steps of the ZZ Method

1. **EOLine**: Orient all edges and solve two opposite cross edges on the bottom (using the M-slice).
2. **F2L**: Solve the First Two Layers efficiently using only R, U, and L moves (no cube rotation needed).
3. **LL**: Use algorithms to orient and permute the Last Layer, typically using OLL and PLL or alternative ZZ-specific variants.

2.2.5 Roux Method

The Roux method is a block-building approach to solving the Rubik's Cube that emphasizes efficiency and minimal rotation. It starts by building a 1×2×3 block on one side of the cube, followed by constructing a second 1×2×3 block on the opposite side. With these two large blocks solved, the method proceeds to orient and position the remaining edges using middle slice moves, which is both move-efficient and ergonomically favourable. Finally, the last step solves and permutes the remaining four corners. Unlike CFOP, Roux does not follow a layered progression and uses fewer algorithms, relying instead on intuition and pattern recognition. Its low move count and freedom of movement make it a powerful method in the hands of advanced cubers, particularly those who favour planning over algorithm memorization. [Spe24b]

2.3 Computational Solutions

For years, humans have tried to find optimal solutions for the Rubik's Cube. They were searching for the so called *God's Number*. In the context of Rubik's Cube, *God's Number* refers to the maximum number of moves required to solve the cube from

any scrambled state, using the fewest possible moves (optimal solution). It has been proven that God's Number for a standard 3x3x3 Rubik's Cube is 20. In Section 2.3 an improvement of the upper bound can be seen during the years.

Date	Lower Bound	Upper Bound	Gap	Notes and Links
July, 1981	18	52	34	Morwen Thistlethwaite proves 52 moves suffice.
December, 1990	18	42	24	Hans Kloosterman improves this to 42 moves.
May, 1992	18	39	21	Michael Reid shows 39 moves is always sufficient.
May, 1992	18	37	19	Dik Winter lowers this to 37 moves just one day later!
January, 1995	18	29	11	Michael Reid cuts the upper bound to 29 moves by analyzing Kociemba's two-phase algorithm.
January, 1995	20	29	9	Michael Reid proves that the "superflip" position requires 20 moves.
December, 2005	20	28	8	Silviu Radu shows that 28 moves is always enough.
April, 2006	20	27	7	Silviu Radu improves his bound to 27 moves.
May, 2007	20	26	6	Dan Kunkle and Gene Cooperman prove 26 moves suffice.
March, 2008	20	25	5	Tomas Rokicki cuts the upper bound to 25 moves.
April, 2008	20	23	3	Tomas Rokicki and John Welborn reduce it to only 23 moves.
August, 2008	20	22	2	Tomas Rokicki and John Welborn continue down to 22 moves.
July, 2010	20	20	0	Rokicki, Kociemba, Davidson, and Dethridge prove God's Number is 20.

Table 2.2: Historical Progress of Upper and Lower Bounds for Solving the Rubik's Cube

2.3.1 Thistlethwaite's Algorithm

Thistlethwaite's algorithm, developed by Morwen Thistlethwaite in 1981, represents a significant milestone in the computational solving of the Rubik's Cube [Sch81]. The method strategically reduces the cube's complexity by dividing the solution into four successive groups of moves, each with increasingly restrictive constraints. This step-by-step reduction guides the cube through nested subgroups, gradually limiting allowable moves while maintaining progress toward a solved state. Initially, this approach brought down the upper bound for solving the cube to 52 moves, a remarkable improvement at the time. Thistlethwaite's method not only demonstrated the power of group theory in practical applications but also paved the way for more refined solving techniques. Its theoretical foundation and structured design have made it a reference point in the evolution of optimal cube-solving strategies.

2.3.2 Kociemba's Algorithm

Kociemba's Algorithm is a two-phase method for optimally solving the Rubik's Cube, significantly reducing the number of moves compared to traditional layer-by-layer methods. It is widely used in computer solvers and speed-solving tools due to its efficiency and ability to find near-optimal solutions, as per [RKDD14].

Overview

The algorithm divides the solution into two distinct phases:

- **Phase 1:** Transforms the cube into a subset of states where all edge orientations are correct, corner orientations are correct, and the cube is in a group known as the *G1* group. This phase ignores the position of the pieces and focuses on orientation constraints.
- **Phase 2:** Solves the cube from a G1 state to the solved state, considering only moves that preserve the constraints achieved in Phase 1. It corrects piece positions without disturbing orientations.

Move Restrictions

Each phase limits the allowed moves to reduce the state space:

- **Phase 1:** All face turns are allowed: U, D, L, R, F, B and their respective primes and double turns.
- **Phase 2:** Only half-turn metric moves are allowed: $U, D, L2, R2, F2, B2$.

Heuristics and Tables

Kociemba's method uses large precomputed lookup tables for pruning the search space:

- **Orientation tables** for edges and corners
- **Permutation tables** for edge slices and corners

These tables help estimate the minimum number of moves to reach the solved state, guiding the solver efficiently through the search tree.

Advantages

- Finds solutions in 20 moves or fewer.
- Significantly faster than brute-force search due to the two-phase approach and pruning.
- Adapted in many solving libraries, such as `Cube Explorer`, `kociemba.py`, and `min2phase`.

Kociemba's Algorithm is an excellent balance between optimality and performance, making it the preferred method in many computer-based cube solvers.

2.3.3 Robotic Cube Solvers

The Rubik's Cube has inspired not only algorithmic research but also mechanical ingenuity through the development of autonomous robotic solvers. These robots integrate vision systems, high-speed motors, and optimal solving algorithms to solve the puzzle with extreme precision and speed.

One of the most iconic examples is **Sub1 Reloaded**, a robot developed by engineer Albert Beer, which set a Guinness World Record in 2018 by solving the cube in just **0.637 seconds** [Rec17]. The robot employed high-performance microcontrollers, computer vision, and Kociemba's two-phase algorithm to analyze and solve the cube faster than any human.

Even more impressively, the current world record for the fastest robot to solve a Rubik's Cube is held by a robot named "Purdubik's Cube" built by Purdue University students. This record was set on April 21, 2025 [Rec25], and is recognized by Guinness World Records. This surpasses the previous record of 0.305 seconds held by Mitsubishi Electric.

Beyond record-breaking prototypes, there are also **commercial robotic solvers** available. One such example is the **GAN Robot**, developed by GAN Cube. This

device pairs with GAN smart cubes and can autonomously scramble or solve the puzzle using a smartphone app [Cub24]. It combines compact hardware with real-time Bluetooth communication and uses embedded solving logic to bring robotic solving into homes and classrooms.

These robots showcase the intersection of software, hardware, and AI-driven logic, offering exciting applications in both educational and entertainment contexts. While some serve as experimental platforms for research and engineering, others make speed-solving accessible to everyday users.

Chapter 3

How to Solve a Rubik's Cube

There are different methods to solve the Rubik's Cube. In this section, a step by step guide for a beginner's method that is used as an introduction into solving the cube is analyzed, as well as the adaptation required for it to be integrated in a computational environment

3.1 Layer by Layer method adapted for Computers

Layer by Layer is a method for beginners that uses a minimum number of algorithms to the whole puzzle [Rom17b]. This is a popular method that makes the transition to more complex methods more natural, building on the already existing knowledge gained from this method.

It relies on seven steps that build on top of each other.

- Step 1: **White Cross**
- Step 2: **White Corners**
- Step 3: **Second Layer**
- Step 4: **Yellow Cross**
- Step 5: **Yellow Corners**
- Step 6: **Permute Yellow Corners**
- Step 7: **Permute Yellow Edges**

A Rubik's Cube has 6 centres that stay fixed and do not change their position with respect to other colours (e.g: the red centre is always surrounded by the yellow, green, white, and blue centres, and opposed to the orange centre). Although the

steps do not impose which colours to select, most of the solvers prefer to start with the white face down.

While many of the algorithms are standard, a computer must perform additional steps for each layer to ensure that cubies are correctly positioned before executing them. While certain moves come naturally to a human solving the puzzle physically, these intuitive interactions require careful handling and explicit configuration when implemented programmatically.

3.1.1 Step 1: White Cross

Many beginners consider this step to be the most challenging part of solving the cube, as it does not rely on a fixed algorithm to complete it automatically. The "cross" (Fig. 3.1) refers to a specific arrangement of the white edge pieces: they are moved to the face with the white centre, aligning their white stickers next to the white centre, while the other colour on each edge matches the centre of the adjacent face.

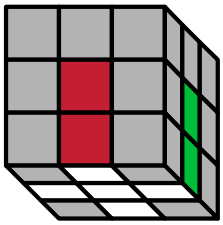

Figure 3.1: Cube with the white cross

The main idea for this step is to bring the edge that contains white on top of the centre of the other colour. From that point on, there are only 2 available cases: either the edge has the cuby with the same colour adjacent to the centre (Fig. 3.2), or the white cuby is adjacent to the coloured centre (Fig. 3.3).

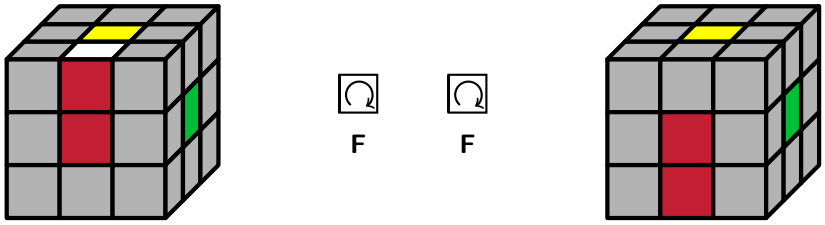

Figure 3.2: Cuby with the same colour adjacent to the center: F2

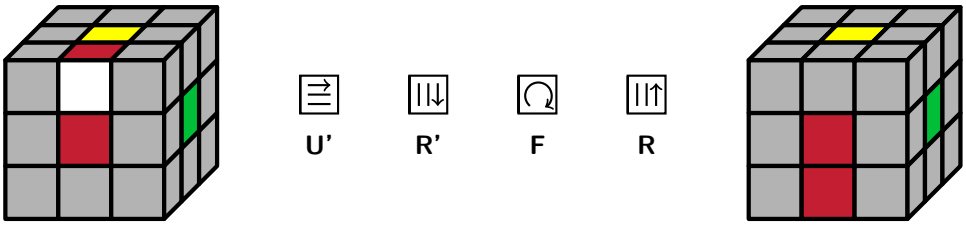

Figure 3.3: Cuby with white adjacent to the center: **U' R' F R**

One could wonder why the last **R** is required. In the case the edge corresponding to green is already placed in the cross, **R'** would just destroy it, so the move is reversed at the end to ensure the cross is still intact.

This step is done for all existing edges. Although it might seem difficult, each edge can be quickly brought on the top face and ensure that the cross is not destroyed at all. This approach is taken in the algorithm used by the application to properly detect where each edge has to be placed.

Obviously, it is not mandatory to bring the edge to the top, as there are scenarios where an edge is on the lateral of the front face, which can be placed on a single move on its position. Bringing it on the top face still represnts a standard way for handling all the scenarios.

For a computer, this is the step that requires the most ingenuity, as it does not have a standard method to solve. As a consequence, from a computational strategy point of view, for solving the white cross, the computer sequentially looks for the edge of the colour of the front and down face, and performs one of the algorithms described. Section 3.1.1 shows all possible white edge positions and their corresponding algorithms. When an edge is inserted, the cube is rotated using a Y move on a clockwise direction. This process is repeated until the white cross is completely solved.

Although some moves might appear unnecessary, the key principle behind this approach is to preserve the edges already placed on the white cross. To achieve this, whenever a move disrupts previously positioned edges, it is followed by countermoves —applied after bringing the target edge to the top— to restore the correct configuration.

Complete White Cross Formation Process

1. Identify a white edge piece that matches the current front face centre colour.
2. Locate the edge's current position using Section 3.1.1.
3. Execute the corresponding positioning algorithm to bring the edge to the bottom layer.
4. Verify that the edge is correctly oriented (white on bottom, matching colour on side).
5. Rotate the cube with a Y-move to position the next target edge as the front-bottom edge.
6. Repeat steps 1-5 until all four white edges form a complete cross on the bottom face.
7. Ensure that each edge's side colour matches the respective face centre before proceeding.

Edge Location	**Position Description**	**Algorithm**	**Moves**
F-D	Front-bottom edge (target)	—	0
F-R	Front-right middle edge	F	1
F-L	Front-left middle edge	F′	1
F-U	Front-top edge	F F	2
U-R	Top-right edge	R′ F R	3
U-L	Top-left edge	L F′ L′	3
U-F	Top-front edge	U L F′ L′	4
U-B	Top-back edge	U′ L F′ L′	4
R-D	Right-bottom edge	R R U F F	5
R-F	Right-front middle edge	R U R′ F F	5
R-B	Right-back middle edge	R′ U R F F	5
R-U	Right-top edge	U F F	3
L-D	Left-bottom edge	L L U′ F F	5
L-F	Left-front middle edge	L′ U′ L F F	5
L-B	Left-back middle edge	L U′ L′ F F	5
L-U	Left-top edge	U′ F F	3
B-D	Back-bottom edge	B B U U F F	6
B-L	Back-left middle edge	L L F′ L L	5
B-R	Back-right middle edge	R R F R R	5
B-U	Back-top edge	U U F F	4
D-F	Bottom-front edge	F F U L F′ L′	6
D-R	Bottom-right edge	R F R′	3
D-L	Bottom-left edge	L′ F′ L	3
D-B	Bottom-back edge	B B U′ L F′ L′	6

Table 3.1: White Edge Positions and Algorithms

3.1.2 Step 2: White Corners

This step aims to complete the white face, shown in Fig. 3.4. For solving it, since the cross is already placed, the next step would be to insert the corners corresponding to the face. A corner consists of 3 colours, and each one belongs to a single place.

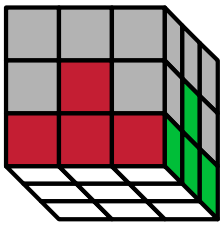

Figure 3.4: Cube with the white face solved

Find a corner in the top layer that contains the white colour. Look at the other colours on that corner as well. For example, the white/red/green corner has been found. This corner will need to be placed between the white-red and white-green edges. Rotate the top layer until the corner is positioned above the place where it needs to go — specifically, in the top layer between the red and green centres. There are three possible cases, described in Fig. 3.5, Fig. 3.6, and Fig. 3.7:

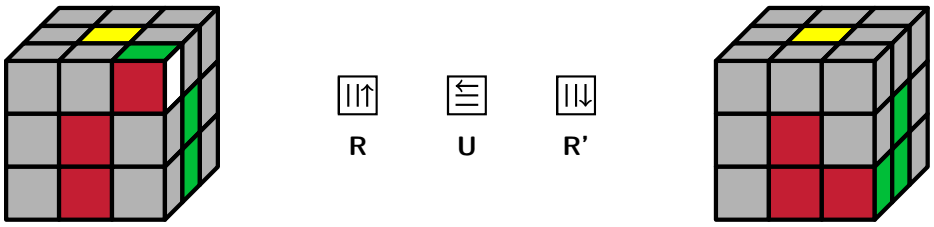

Figure 3.5: **R U R'**

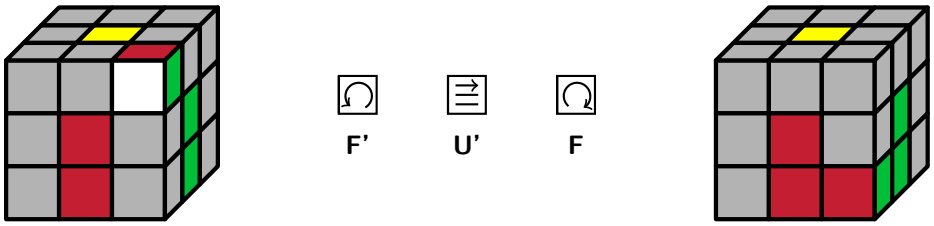

Figure 3.6: **F' U' F**

Figure 3.7: **R U U R' U' R U R'**

There may also be a situation where the corner is already on the bottom layer, exactly where it needs to go, but with the white facing either the front or the right. In this case, any of the previous algorithms can be applied — preferably **R U R'** because it is the simplest to perform. This will insert another corner in its place, bringing it back up to the top layer. Now, there should be one of the first three cases.

Just like in the first step where the edges were solved (the cross), here this step is applied for all 4 corners so that in the end the cube is brought to the state shown at Fig. 3.4.

Before applying the specific algorithms mentioned in this section, the corners have to be placed in front of the solver. Each corner position has a specific algorithm sequence to bring it to the correct position. The white corner insertion follows a systematic approach:

Complete Corner Insertion Process

1. Locate the white corner with matching front and right centre colours.
2. Execute the positioning algorithm from Section 3.1.2.
3. Apply the orientation algorithms described in this chapter.
4. Rotate the cube with a Y-move and repeat for the next corner.
5. Continue until all four white corners are correctly placed and oriented.

The total corner insertion step typically requires 16-24 moves (4 corners × 4-6 moves each), completing the entire white face and ensuring the first layer is fully solved. Section 3.1.2 shows all possible white corner positions and their corresponding algorithms.

Corner Location	Position Description	Algorithm	Moves
F-R-U	Front-right-top (target)	—	0
F-R-D	Front-right-bottom	R U R′ U′	4
B-R-U	Back-right-top	U	1
B-L-U	Back-left-top	U U	2
F-L-U	Front-left-top	U′	1
F-L-D	Front-left-bottom	L′ U′ L	3
B-L-D	Back-left-bottom	L U U L′	4
B-R-D	Back-right-bottom	R′ U R U	4

Table 3.2: White Corner Positions and Algorithms

3.1.3 Step 3: Second Layer

The next step is to look for an edge in the top layer that does not contain the colour of the top face (yellow) on either side. These are the red-green, green-orange, orange-blue, and blue-red edges.

Next, the cube is held with white on the bottom and red in front. Let's assume an edge that contains red was fount. The top layer (U) is rotated until the red colour of the edge matches the corresponding centre. There are two possible cases, illustrated in Fig. 3.8 and Fig. 3.9:

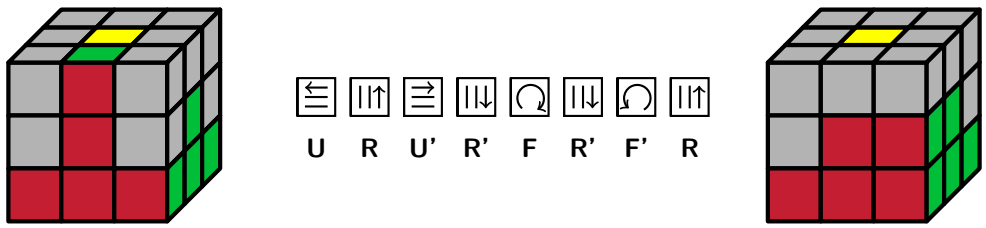

Figure 3.8: **U R U' R' F R' F' R**

Figure 3.9: **y U' L' U L F' L F L'**

There are the only two algorithms required for this step. They are very intuitive.

There might be the situation when the desired edge is already place where it should be, but the sides are switched (Fig. 3.10). In this situation, any of the following algorithms can be applied, so the edge is brought back on the upper face, and after that placed properly.

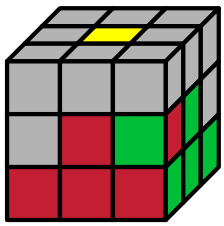

Figure 3.10: Red-Greeen edge is placed properly, but the sides are switched.

Just like in the previous steps, these algorithms are applied for all the 4 edges so that in the end the cube should have the state shown in Fig. 3.11:

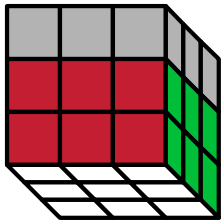

Figure 3.11: First 2 layers completed

As in previous steps, the application follows the same logic used for solving the white cross. At each stage, the focus is on the edge that matches the current face and its corresponding right centre colour. Although there are only 8 possible edge positions, this phase is the most computationally intensive due to the length and complexity of the required algorithms. Once the target edge is brought to the top layer using U moves, it is aligned correctly, and one of the predefined algorithms is applied to insert it properly.

Section 3.1.3 shows all possible middle edges positions and their corresponding algorithms.

Edge Location	Position Description	Algorithm	Moves
F-U	Front-top edge (target)	—	0
R-U	Right-top edge	U	1
B-U	Back-top edge	U U	2
L-U	Left-top edge	U′	1
F-R	Front-right middle edge	U R U′ R′ U′ F′ U F U U	10
F-L	Front-left middle edge	U′ L′ U L U F U′ F′ U U	10
B-R	Back-right middle edge	Y U R U′ R′ U′ F′ U F U′ Y′	11
B-L	Back-left middle edge	Y′ U′ L′ U L U F U′ F′ U Y	11

Table 3.3: Middle Edge Positions and Algorithms

Complete Middle Edge Solving Process

1. Locate the middle edge with matching front and right centre colours in the top layer.
2. If the edge is not in the top layer, remove it using the appropriate algorithm from Section 3.1.3.
3. Position the edge above its target location using U moves.
4. Determine the edge orientation: if front colour shows on front face, use right-hand algorithm; otherwise use left-hand algorithm.
5. Execute the corresponding insertion algorithm to place the edge in the middle layer.
6. Rotate the cube with a Y-move and repeat for the next middle edge.
7. Continue until all four middle edges are correctly positioned, completing the first two layers.

3.1.4 Step 4: Yellow Cross

In this step, the edges of the top layer are oriented. Simply put, a yellow cross is made. This is a very easy step with two very simple and elegant algorithms. The most important thing here is to understand how to recognize the cases, and the key is that corners do not represent any interest at this moment.

At this step, it is important how the edges that contain yellow align. There are 4 cases:

- **Yellow Cross is already completed**: Step marked as complete.
- **A yellow line is visible (Fig. 3.12)**: Turn the top layer (**U** turns) until the line is placed like in the figure below, then apply the algorithm for the line.
- **A yellow L is visible (Fig. 3.13)**: Turn the top layer (**U** turns) until the L is placed like in the figure below, then apply the algorithm for the L.
- **Only the centre has yellow (Fig. 3.14)**: Apply any of the algorithms one, then detect the new case (either a line or an L), and apply the corresponding algorithm.

Figure 3.12: Yellow line: **F R U R' U' F'**

Figure 3.13: Yellow L: **F U R U' R' F'**

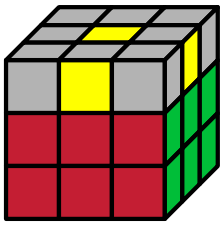

Figure 3.14: Just the centre is yellow

The application does not need anything auxiliary for having this step solved.

3.1.5 Step 5: Yellow Corners

The only remaining pieces to have the yellow face solved are the corners. A piece that is already oriented is a happy case. It might be necessary to orient 2, 3 or 4 corners. It is impossible to have only one corner to orient.

There are 2 possible states for each corner at this time

- **Yellow corner is on the front face** (Fig. 3.15)
- **Yellow corner is on the right face** (Fig. 3.16)

Figure 3.15: Yellow corner on the front face: **R' D R D' R' D R D'**

Figure 3.16: Yellow corner on the front face: **D R' D' R D R' D' R**

After applying any of the algorithms one, the cube will be scrambled. This is something expected, but the face that the solver views must not be changed. Accessing the other corners should be done only by applying **U** moves.

For each corner, the above step has to be done. After that, the cube should look like in Fig. 3.17.

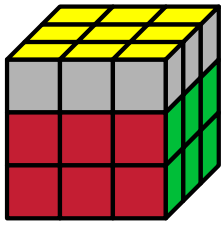

Figure 3.17: Rubik's cube after the 5th step

As in the previous step, the application does not have any auxiliary step required for solving it.

Complete Yellow Corner Orientation Process

1. Position a corner that needs orientation in the front-right position using U moves.
2. Identify the location of the yellow sticker on the corner piece.
3. If yellow sticker is on the front face, apply: R' D R D' R' D R D'.
4. If yellow sticker is on the right face, apply: D R' D' R D R' D' R.
5. A yellow sticker is on top, rotate to the next corner with U.
6. Repeat steps 1-5 until all four corners show yellow on the top face.
7. Verify that all corner pieces have their yellow stickers oriented upward.

3.1.6 Step 6: Permute Yellow Corners

As the yellow face is solved, the remaining steps are related to permuting the edges and the corners. This step focuses on the corners. If all of them were properly placed, this step should be skipped.

Most of the time, there is a case where two corners on one face of the top layer share the same colour, while the other faces do not, as in Fig. 3.18.

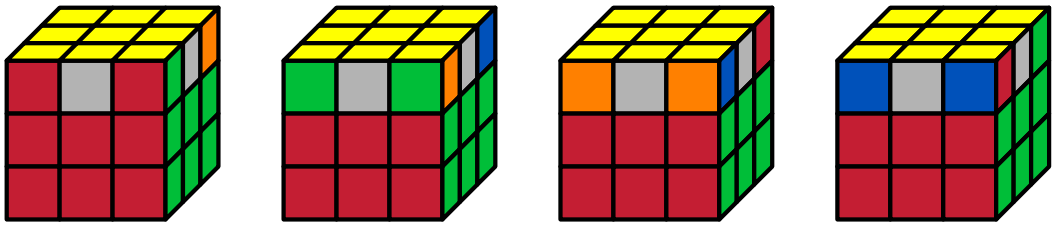

Figure 3.18: Two aligned corners

Before applying the algorithm, the face with the 2 corners must face the solver, to be on the front. After that, the following algorithm written at Fig. 3.19 has to be applied:

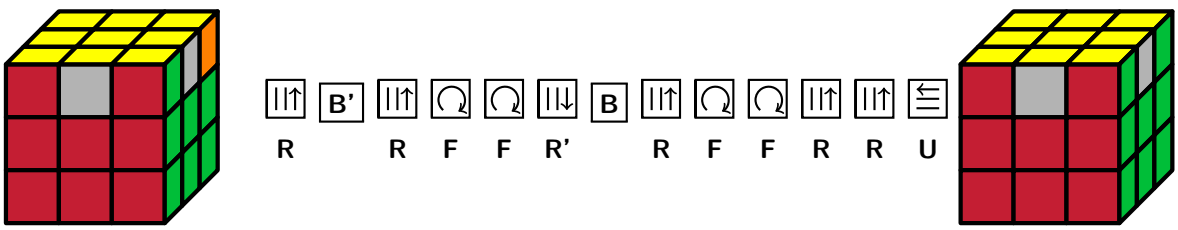

Figure 3.19: Permute Yellow Corners: **R B' R F F R' B R F F R R U**

If initially no face has two corners of the same colour, applying the algorithm once will cause two matching corners to appear on one of the cube's faces.

3.1.7 Step 7: Permute Yellow Edges

The last step for solving the cube is to permute the edges from the top face. From now on, a single algorithm is required. At this step, three different cases can be encountered, but all of them can be solved using the same formula.

When one side is already complete and only three edges still need to be corrected, the completed side is placed at the back (not facing the solver). From this position, the algorithm will always be applied. At Fig. 3.20, the completed side is orange, but it can be any other colour.

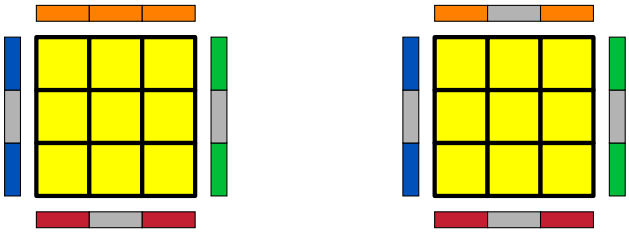

Figure 3.20: The cases for permuting yellow edges

The only algorithm required is

For Case 1, if the cube isn't solved after the first application, the algorithm is applied once more. Sometimes, it may need to be applied twice.

For Case 2, when no side is solved and all four edges need to be corrected, the algorithm is applied once from any position. After that, one side will be completed, we move it to the back, and the new state should correspond to the first case.

Now, the whole cube is solved.

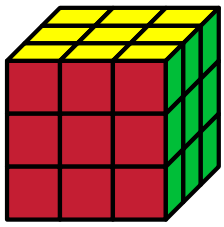

Figure 3.21: Solved puzzle

Chapter 4

Component Detection

4.1 Detecting colours

In addition to shape detection, accurately identifying the colour of each cuby is essential. A standard Rubik's Cube consists of six distinct colours: red, orange, yellow, white, green, and blue. Each of these colours fall within a specific range, allowing for their estimation based on predefined thresholds.

To facilitate accurate colour matching, the **Hue, Saturation, and Value (HSV)** colour model is integrated, providing a robust representation for distinguishing between colours under varying lighting conditions.

4.1.1 Hue, Saturation, and Value

In the field of image processing, the RGB (Red, Green, Blue) colour model is among the most widely adopted representations. Within this model, any colour can be expressed as a linear combination of three primary components: red, green, and blue. Despite its widespread usage, the RGB model is notably sensitive to variations in lighting conditions, often resulting in differing colour values for the same object under varying illumination. This sensitivity can significantly complicate colour detection and analysis tasks.

Hue, Saturation, and Value (HSV) is a cylindrical-coordinate representation of points in an RGB Colour model. The **HSV (Hue, Saturation, Value)** colour model describes colours in a way that more accurately represents human perception. HSV splits colours into 3 main components, unlike RGB which aims to mix primary colours.

The three components (see Fig. 4.1) that describe HSV are:

- **Hue**: Represents the type of the colours (base colour)
- **Saturation**: Represents the intensity of the colours.

- **Value**: Represents the brightness of the colours.

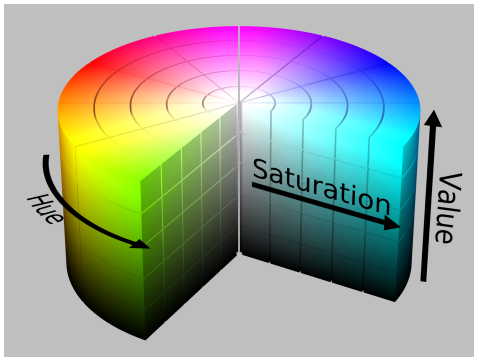

Figure 4.1: HSV colour space represented as a solid cylinder.

Inside the cylinder, the angle around the central vertical axis maps the hue, the distance from the main axis corresponds to saturation, and the distance along the axis corresponds to value.

The algorithm uses the HSV (Hue, Saturation, Value) colour space for colour detection. HSV is preferred over RGB because:

- It separates colour information (Hue) from brightness (Value)
- It's more robust to lighting variations
- It better matches human colour perception

When a pixel value within the RGB colour space is adjusted, the intensities of its red, green, and blue channels are simultaneously altered. Consequently, attributes such as colour, brightness, and saturation are entangled, making it challenging to distinguish between them in complex visual environments. To address this limitation, the HSV (Hue, Saturation, Value) colour space is frequently employed. This model decomposes colour information into three distinct components, hue, saturation, and value, allowing a more independent and intuitive interpretation of colour variation. [CYCT10]

As described above, the HSV colour space offers a more robust framework for analysing colour, intensity, and brightness, particularly in environments where lighting conditions are variable. The transformation from RGB to HSV is defined by the following equations:

$$
\begin{aligned}
h &= \begin{cases} 0, & \text{if } \max = \min \\ 60^\circ \times \left(\frac{g-b}{\max - \min} \bmod 6\right), & \text{if } \max = r \\ 60^\circ \times \left(\frac{b-r}{\max - \min} + 2\right), & \text{if } \max = g \\ 60^\circ \times \left(\frac{r-g}{\max - \min} + 4\right), & \text{if } \max = b \end{cases} \\
s &= \begin{cases} 0, & \text{if } \max = 0 \\ \frac{\max - \min}{\max}, & \text{otherwise} \end{cases} \\
v &= \max
\end{aligned}
\tag{4.1}
$$

where r, g and b represent red, green and blue values normalised in value $[0, 1]$

4.1.2 Image analysis

Although the size of each cuby remains consistent throughout the puzzle, the colour of each individual piece must be analysed separately. Fortunately, the limited and predefined set of colours on a standard Rubik's Cube allows for individual assessment of each colour. Adhering to the original colour scheme of the Rubik's Cube, each colour is associated with a specific interval that defines the range within which it can be accurately identified. The corresponding values are attached in Table 4.1

Colour	HSV Minimum (H, S, V)	HSV Maximum (H, S, V)
Red	(0, 101, 0) or (170, 101, 0)	(5, 255, 255) or (180, 255, 255)
Orange	(6, 101, 0)	(10, 255, 255)
Yellow	(11, 101, 0)	(35, 255, 255)
Green	(36, 101, 0)	(75, 255, 255)
Blue	(76, 101, 0)	(130, 255, 255)
White	(0, 0, 0)	(180, 100, 255)

Table 4.1: HSV Value Ranges for Rubik's Cube Colours

The classification of Rubik's Cube colours based on HSV ranges presents specific challenges for each group. Red and orange, due to their proximity on the hue spectrum, often suffer from overlapping hue values, especially under varying lighting conditions or camera white balance issues. This makes distinguishing between the two particularly difficult, as slight shifts in saturation or brightness can cause misclassification.

Similarly, the yellow and green group also poses a challenge, particularly in environments with reflective surfaces or strong ambient light. The hue range between yellow and green is narrower, and under uneven illumination, green cubies may appear yellowish or vice versa.

Blue generally stands out more distinctly in the HSV space; however, it may still be affected by shadows or poor lighting, which can reduce saturation and shift its hue toward other colours, such as purple or cyan.

Lastly, white is especially sensitive to brightness and saturation. Since it occupies a broad hue range but requires low saturation and high brightness, it is easily misidentified when lighting conditions are suboptimal, leading to confusion with pale versions of other colours. These challenges necessitate careful calibration and potential preprocessing techniques to ensure robust colour detection. As the range that covers the hue for white covers a considerable part of the spectrum, a face of a cuby that can not be properly identified for any of the other 5 colours will be assumed to be white.

Colour Isolation

Clearly, a cuby should only be active for a single colour. To properly analyse these, each individual colour of interest has to be isolated. To extract colours, in computer vision, **bitmasks** are used.

In OpenCV, a bitmask is a binary image used to isolate specific regions of interest within an image, based on certain criteria such as colour range. When determining the colour of a cuby on a Rubik's Cube, bitmasking proves to be a highly effective technique. By applying colour thresholding within the HSV colour space, a binary mask is generated, where pixels that fall within a specified HSV range are usually marked as white (value 255) - but any other colour can be used, for different purposes - and all others as black (value 0).

This mask allows for focused analysis by filtering out irrelevant areas and isolating only those pixels that potentially match the target colour. Once the mask is obtained, additional operations such as contour detection, pixel counting, or mean colour calculation can be applied to validate and refine the cuby's colour classification. This approach is particularly useful in scenarios involving complex backgrounds or inconsistent lighting, as it ensures that only the relevant region is evaluated for colour identification.

Chapter 5

Web application

5.1 Requirements and Specifications

The proposed system is composed of two principal components, each responsible for distinct functionalities essential to the overall application:

- **User Management:** Handles user authentication and authorization for accessing the platform.
- **Image Processing and Solution Generation:** Processes Rubik's Cube images to determine the correct sequence of moves required to solve the puzzle.

System Architecture and Technology Stack

The user-facing portion of the application will be developed using **Next.js**, a full-stack React framework that offers serverless capabilities and streamlines the deployment process [Ver25]. Authentication will be implemented through **NextAuth.js**, enabling seamless integration with third-party providers such as Google or Facebook [Col25]. This approach not only enhances the security and scalability of the system but also simplifies user account management.

The backend services will be powered by **Flask**, a high-performance Python web framework, well-suited for handling real-time image processing tasks. This API will be responsible for analysing the cube's face images and computing the solution steps based on the determined colour configuration. [Pal10]

As architecture for the frontend, there are some design pattern that come in handy for the intended use case, that will be detailed in the following chapters. A high level overview of the proposed design can be observed in Fig. 5.1.

Functional Requirements

- The system shall allow users to log in using third-party providers (e.g., Google).

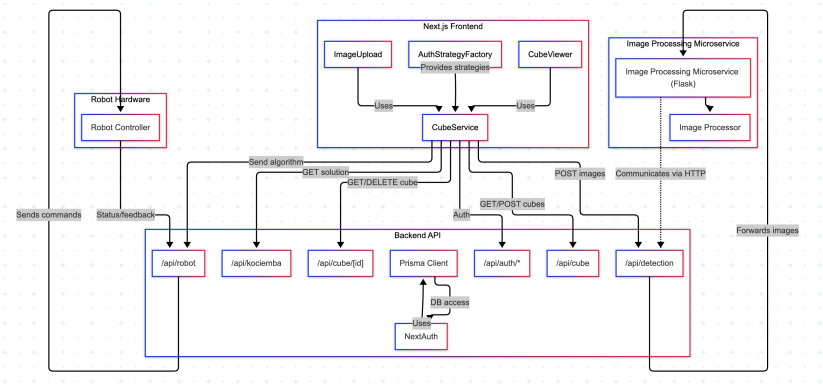

Figure 5.1: High-level architecture of the application.

- The system shall accept image uploads for each of the six faces of a Rubik's Cube, either from live camera input or local storage.
- The server shall process uploaded images to extract colour data and reconstruct the cube's current state.
- The application shall display a real-time 3D rendering of the cube based on detected colours.
- The system shall compute and return a step-by-step solving sequence for the cube.
- The user shall be able to choose between a "fastest solution" mode and a "complete solving sequence" mode.
- The backend shall respond to requests using JSON format and standard HTTP status codes.

Non-Functional Requirements

- The system shall provide consistent performance and accuracy in colour detection under standard lighting conditions.
- The backend shall be scalable and capable of handling concurrent image processing requests.
- Communication between frontend and backend shall adhere to REST principles, using form-encoded request bodies.
- The application shall maintain cross-platform compatibility across desktop and mobile devices.

Workflow Overview

When initiating a new solve, the user uploads images of the six cube faces. These images are analysed on the server, and a colour mapping is generated. A 3D model of the cube is rendered client-side to visually represent the current state. Following analysis, the server computes the solving sequence and returns it to the client. The user can then view the required steps, depending on their preference for a minimal or comprehensive solution path.

5.2 Frontend

5.2.1 Use Case Specification

Composed from multiple parts, the application serves to the end user as a way to solve a Rubik's cube manually on a web interface, while as well to be guided on how to solve and learn the required steps for any possible scramble.

A solver should be able to perform any desired moves on a cube, either by manually inputing each step, or by performing an algorithm by a single press of the button.

For linking the real representation of the cube to a virtual environment, an user can also scan its cube by providing 6 images, one for each face. A response from the backend is send upon the request, returning the virtual representation of the cube.

5.2.2 User Interface

The user is welcomed by a hero section, with a message displayed to emphasize the end goal of the application, as shown in Fig. 5.2. From there, 2 options are provided:

- For user that are not logged in, connect via social authentication
- For user that are logged in, visit the online solver of the puzzle

5.2.3 User Authentication

The core system is built on NextAuth, an open-source authentication solution designed for Next.js applications. It serves as the central hub connecting various authentication methods, such as traditional email-password logins or social media providers. Internally, it uses a predefined user model in the database that follows a specific structure. For each supported authentication method, NextAuth offers a tailored strategy, using distinct credentials and configurations to handle each type of social media login.

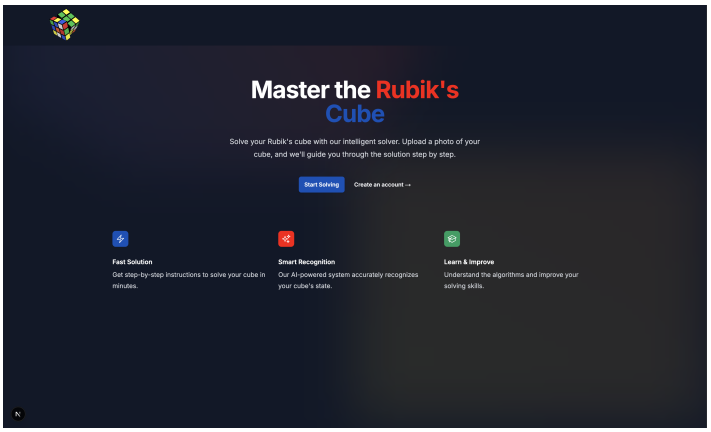

Figure 5.2: Welcome menu of the web interface.

In this application, the methods used are based on Gmail and Facebook login. The user just has to click on the button, then it is redirected to the account of the social media platform, and get the approvals to login to the app.

5.2.4 The cube

A cube is treated as an object, being an unique instance across the whole application. Each cube is represented as a composition of 6 3×3 matrices, corresponding to each one of the faces. The key for each face is associated to the faces used for each move, with respect to the red face on the front, and the yellow face on top.

Physical Movement

Each move is conceptualized as an individual object responsible for applying a specific transformation to the cube's state. This object-oriented approach is particularly effective for maintaining and managing the cube's previous states, as it adheres to the principles of the Command design pattern. An abstract base class establishes a default method, which must be concretely implemented by all subclasses, ensuring consistent behaviour across different move types.

Notably, every move possesses an inherent countermove, realized by applying the same operation in the reverse direction. To facilitate this functionality, each move class incorporates a boolean attribute that denotes the direction of the movement. The undo method systematically reverses the preceding operation by invoking its inverse, thereby restoring the prior state of the cube.

Moreover, since an algorithm is represented as a sequence of individual moves, the overall structure exhibits a tree-like architecture, making it suitable for the application of the Composite design pattern. The algorithm itself extends the abstract base class for moves, while additionally maintaining a collection of constituent moves

that are executed sequentially whenever the algorithm is invoked. This same structural principle is applied to the undo functionality, whereby the algorithm systematically reverses each move in the sequence to restore the prior state.

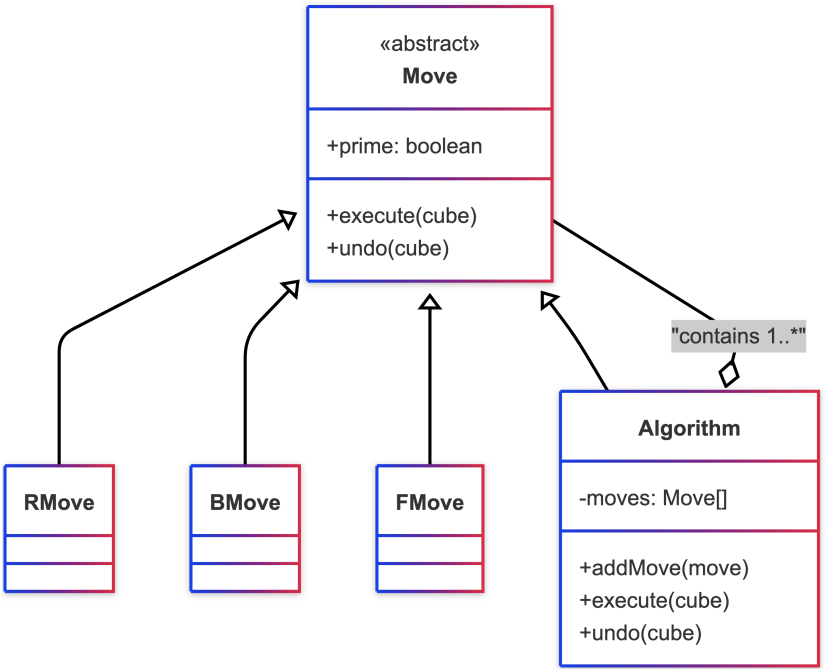

Figure 5.3: Diagram for Moves and Algorithms.

The Rubik's cube moves follow a consistent pattern where each move rotates one face and cycles elements between four adjacent faces. Table Section 5.2.4 shows the basic transformations.

Move	Effect
R	Rotate right face clockwise, cycle U→F→D→B edges
R′	Rotate right face counter-clockwise, cycle U→B→D→F edges
U	Rotate up face clockwise, cycle F→L→B→R top rows
U′	Rotate up face counter-clockwise, cycle F→R→B→L top rows
F	Rotate front face clockwise, cycle U→R→D→L adjacent edges
F′	Rotate front face counter-clockwise, cycle U→L→D→R adjacent edges
L	Rotate left face clockwise, cycle U→B→D→F edges
L′	Rotate left face counter-clockwise, cycle U→F→D→B edges
B	Rotate back face clockwise, cycle U→L→D→R adjacent edges
B′	Rotate back face counter-clockwise, cycle U→R→D→L adjacent edges
D	Rotate down face clockwise, cycle F→R→B→L bottom rows
D′	Rotate down face counter-clockwise, cycle F→L→B→R bottom rows

Table 5.1: Basic Cube Moves

Face Rotation Mechanics

When a face rotates 90° clockwise, the 3×3 face matrix follows a distinct transformation pattern:

Table 5.2: **Face Rotation Pattern (Clockwise and Counter-clockwise)**

Original	Clockwise	Counter-clockwise
$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	$\begin{bmatrix} 7 & 4 & 1 \\ 8 & 5 & 2 \\ 9 & 6 & 3 \end{bmatrix}$	$\begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 8 \\ 1 & 4 & 7 \end{bmatrix}$

Key Properties of Face Rotations

- Each move affects exactly **20 stickers**: 9 on the rotating face + 12 on adjacent faces.
- **Prime moves** (e.g., R', U') are the exact inverse of their respective base moves.
- All moves **preserve the group structure** of the cube and can be composed.
- **Center pieces never move**, maintaining the fixed identity of each face.

Cube representation: 2D

The cube is represented on a flat area, displayed as a cross for the user. Each move has an associated button that performs the corresponding transition of the state of the cube, as shown in Fig. 5.4

Beneath the flat (2D) representation of the cube, there are seven buttons representing the main sequential steps needed to solve the cube. These buttons control the algorithm steps for both the 2D and 3D visualizations.

Cube representation: 3D

The application features an interactive 3D visualization of the Rubik's Cube, built using React Three Fiber—a React-based renderer for Three.js that allows for declarative 3D scene management. This setup enables users to rotate and inspect the cube from various angles, with real-time synchronization to the cube's internal state, as

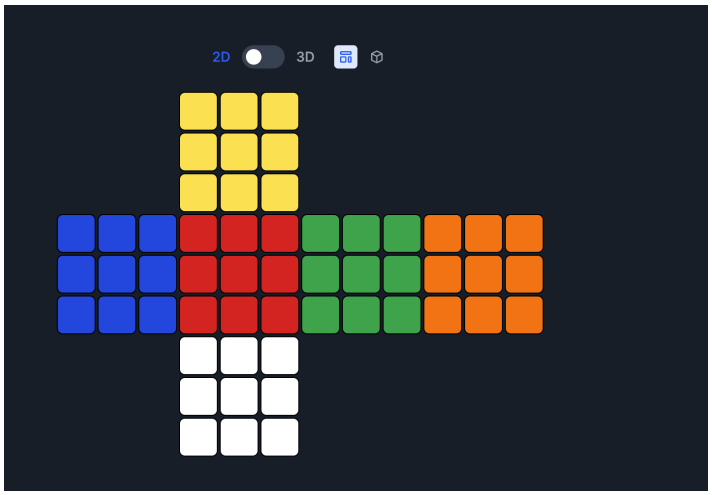

Figure 5.4: 2D Cube representation.

displayed in Fig. 5.5. A toggle button allows users to seamlessly switch between 2D and 3D representations while preserving the current state of the cube.

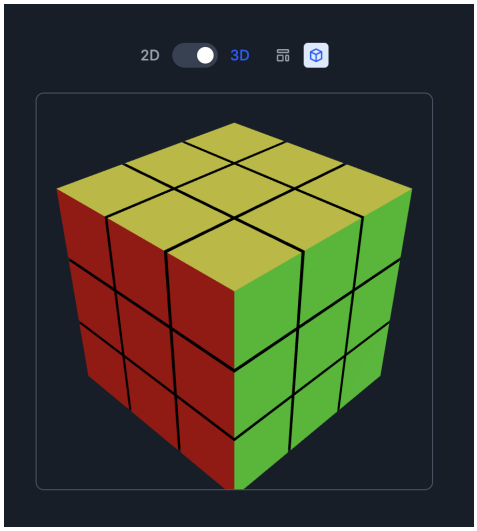

Figure 5.5: 3D Cube rendering.

The 3D rendering system is built upon several key technologies:

- **React Three Fiber (@react-three/fiber)**: A React reconciler for Three.js that enables declarative 3D scene construction using JSX syntax [Poi25b]
- **Drei (@react-three/drei)**: A collection of useful helpers and abstractions for React Three Fiber, specifically utilized for camera controls [Poi25a]
- **Three.js**: The underlying WebGL library providing the 3D rendering capabilities [mrd25]

The 3D cube is rendered as a collection of 27 individual cubies arranged in a

$3 \times 3 \times 3$ configuration, where each cuby represents one of the smaller cubes that compose the complete Rubik's cube.

Each cuby is implemented as a `Cuby` React component that renders a Three.js mesh with box geometry. The cuby dimensions are set to $0.95 \times 0.95 \times 0.95$ units, creating small gaps between adjacent cubies that enhance visual separation and provide a more realistic appearance.

Colour Mapping System and Coordinate Transformation

The colour mapping system converts the logical cube state into a visual representation through a coordinate transformation process:

$$\text{Front face (z = 1):} \quad \text{row} = 1 - y, \quad \text{col} = 1 + x \tag{5.1}$$

$$\text{Back face (z = -1):} \quad \text{row} = 1 - y, \quad \text{col} = 1 - x \tag{5.2}$$

$$\text{Right face (x = 1):} \quad \text{row} = 1 - y, \quad \text{col} = 1 - z \tag{5.3}$$

$$\text{Left face (x = -1):} \quad \text{row} = 1 - y, \quad \text{col} = 1 + z \tag{5.4}$$

$$\text{Top face (y = 1):} \quad \text{row} = 1 + z, \quad \text{col} = 1 + x \tag{5.5}$$

$$\text{Bottom face (y = -1):} \quad \text{row} = 1 - z, \quad \text{col} = 1 + x \tag{5.6}$$

This coordinate transformation ensures proper orientation alignment between the 3D visualization and the logical cube representation, accounting for the different viewing perspectives of each face.

Each cuby utilizes six `meshStandardMaterial` instances, one for each face of the cube. The material assignment follows Three.js conventions where materials are attached to specific faces using the `attach` property:

- Material 0: Right face (+X direction)
- Material 1: Left face (-X direction)
- Material 2: Top face (+Y direction)
- Material 3: Bottom face (-Y direction)
- Material 4: Front face (+Z direction)
- Material 5: Back face (-Z direction)

The colour mapping function `getColorHex()` converts semantic colour names to hexadecimal values:

Colour Name	Hex Value
White	`#ffffff`
Yellow	`#ffff00`
Red	`#ff0000`
Orange	`#ff8000`
Blue	`#0000ff`
Green	`#00ff00`

The 3D rendering system incorporates several interactive features:

Orbit Controls: Implemented using Drei's `OrbitControls` component, allowing users to rotate, zoom, and pan around the cube. The controls include damping for smooth interaction and constraints on minimum/maximum zoom distances (3-20 units) and polar angles.

Click Detection: Each cuby responds to mouse click events through Three.js's raycasting system. The click handler determines which face was clicked by analyzing the `faceIndex` property and mapping it to the corresponding cube face using the cuby's 3D position.

Lighting System: The scene employs a three-light setup for optimal visual quality:

- Ambient light (intensity: 0.6) providing overall illumination
- Directional light (position: [10, 10, 5], intensity: 1.0) simulating sunlight
- Point light (position: [-10, -10, -5], intensity: 0.5) providing fill lighting

Performance Considerations

The 3D rendering system is optimized for performance through several techniques:

- **Component Memoization**: Individual cubies are memoized to prevent unnecessary re-renders.
- **Efficient State Management**: The cube state is synchronized between 2D and 3D views through a unified state management system.
- **Minimal Geometry**: Each cuby uses simple box geometry to minimize rendering overhead.
- **Material Reuse**: Standard materials are used consistently across all cubies.

The 3D visualization integrates seamlessly with the application's cube manipulation features, automatically updating when moves are executed and providing visual feedback for highlighted cubies during step-by-step solving processes.

Cube configuration

To configure the cube, the user can access a dedicated menu that allows uploading six images—one for each face of the cube—to extract the corresponding scramble. Each image is automatically associated with a specific face. Users can either upload a pre-existing photo or capture a new one using the webcam, as shown in Fig. 5.6.

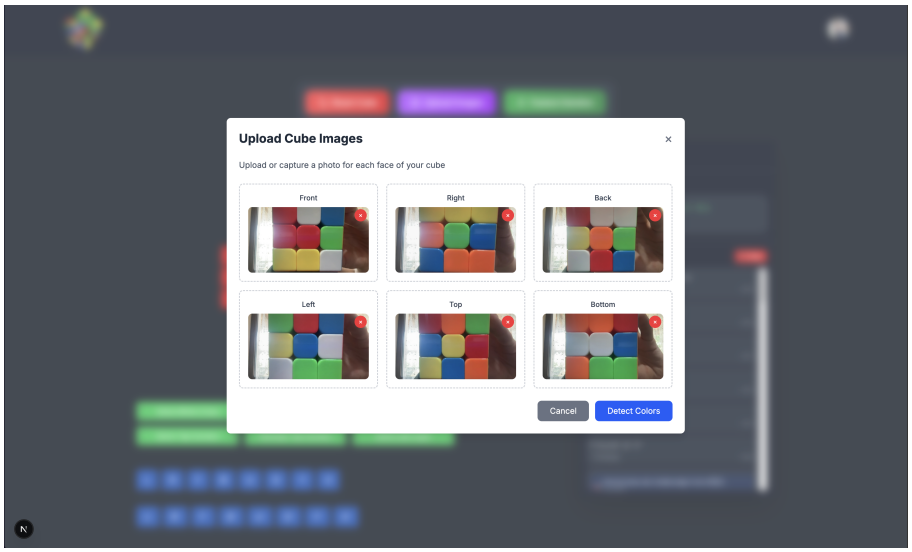

Figure 5.6: Upload menu.

To ensure accurate detection of the cubies, a designated region of interest is displayed on the screen, indicating where the cube should be positioned during image capture (Fig. 5.7).

If image upload is not possible or the detection results are inaccurate, the user can manually configure each cuby by entering the corresponding letter for the face colour. This functionality is available for both 2D and 3D representations. To access the manual configuration menu, the user simply clicks on any cuby of the desired face, triggering the menu to appear (see Fig. 5.8). Once the configuration is saved, the cube's state is immediately updated and reflected on the screen.

Cube control

The user interface provides two main ways to interact with the cube solver (see Fig. 5.9:

- **Step-by-step solving:** The user can click on one of the seven green buttons

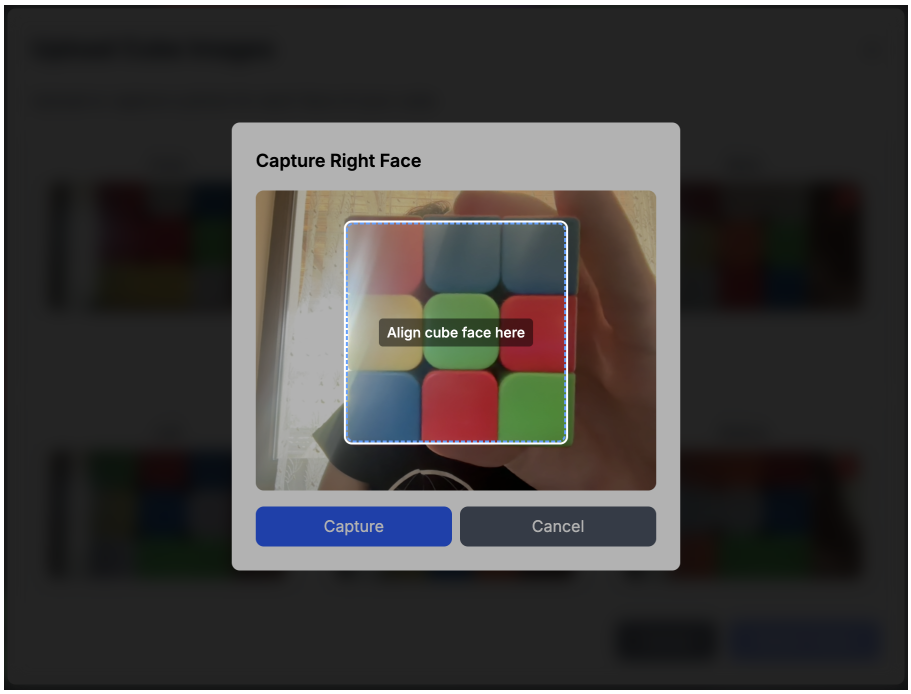

Figure 5.7: Camera with the region of interest.

below the cube, each corresponding to a specific stage in the standard solving process:

  - `Solve White Cross`
  - `Insert White Corners`
  - `Solve Middle Edge`
  - `Make Top Cross`
  - `Solve Top Corners`
  - `Permute Top Corners`
  - `Solve Last Layer`

  Each button applies a predefined algorithm and updates the cube's state accordingly.

- **Manual move execution:** Below the solving buttons, users can also perform individual moves by clicking on any of the 18 blue move buttons representing standard cube notations (e.g., `L`, `R'`, `U2`, etc.). This is useful for custom manipulations or replicating scrambles.

At the top of the interface, additional actions are available:

- **Reset Cube:** Resets the cube to its default state.

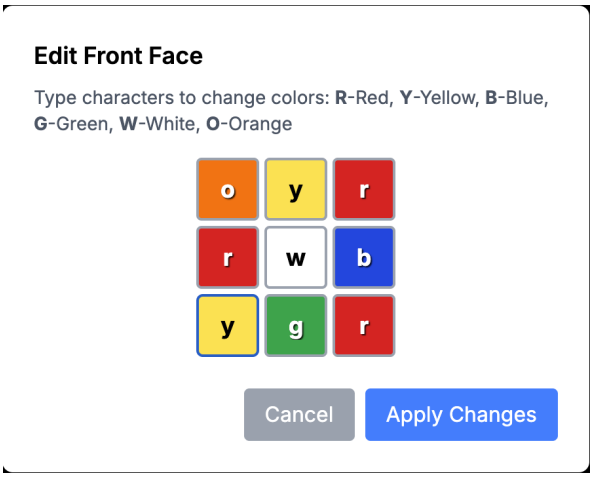

Figure 5.8: Configuration menu for a face.

- **Upload Images:** Allows configuring the cube by uploading six images (one per face).
- **Fastest Solution:** Computes the optimal solution using Kociemba's algorithm.

On the right side of the screen, regardless of whether the 2D or 3D view is selected, a panel labelled **Algorithms** is displayed. This section contains:

- **Current Algorithm:** Shows the algorithm currently being applied or under execution.
- **History:** A chronological list of all previously applied algorithms, allowing users to review and track the steps taken to solve the cube.

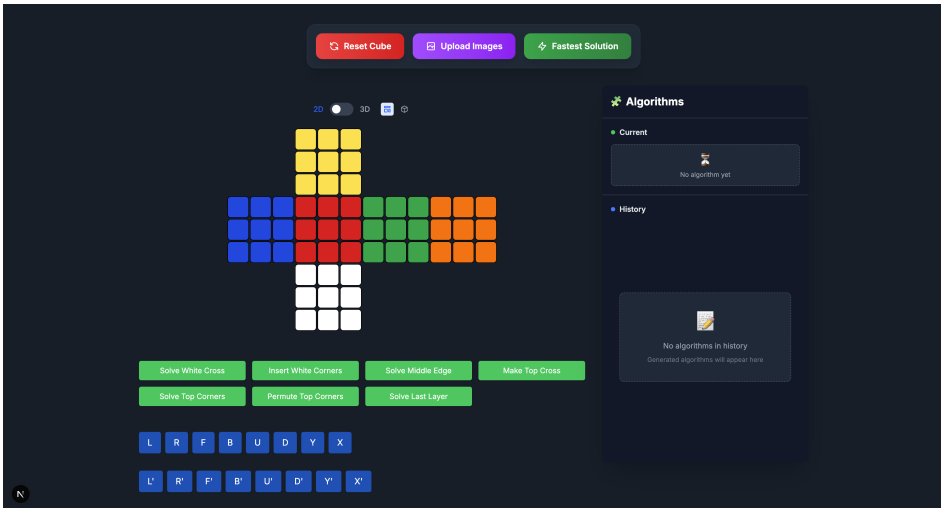

Figure 5.9: Controls for the cube.

5.3 Database

The database schema for this project is designed to efficiently manage users, their authentication sessions, accounts from external providers, and the storage of Rubik's Cube scrambles. The schema is implemented using Prisma, an open-source next-generation ORM (Object-Relational Mapping) tool for Node.js and TypeScript [Pri25]. Prisma provides a type-safe and intuitive API for database access, automates migrations, and integrates seamlessly with PostgreSQL, which is the database provider used in this project.

5.3.1 Entities and Relationships

The main entities in the schema are `User`, `Account`, `Session`, and `Cube`. Their relationships are illustrated in Figure Fig. 5.10.

- **User**: Represents an individual user of the application. Each user has a unique identifier, optional name, email, profile image, and a role (defaulting to "user"). The `User` entity is central to the schema and is related to all other main entities.
- **Account**: Stores information about external authentication providers (such as Google or Facebook) linked to a user. Each account references a user via a foreign key (`userId`). A user can have multiple accounts, but each account is uniquely identified by the combination of provider and provider account ID.
- **Session**: Represents an active authentication session for a user. Each session is linked to a user via `userId` and contains a unique session token and an expiration timestamp. A user can have multiple sessions, for example, if logged in from multiple devices.
- **Cube**: Stores information about Rubik's Cube scrambles saved by users. Each cube record contains a unique identifier, a reference to the user who created it, an optional description, the scramble string, and a creation timestamp. This allows users to save and manage multiple cube states.

The relationships between these entities are as follows:

- A `User` can have multiple `Account` records (one-to-many).
- A `User` can have multiple `Session` records (one-to-many).
- A `User` can have multiple `Cube` records (one-to-many).
- Each `Account`, `Session`, and `Cube` record belongs to exactly one `User`.

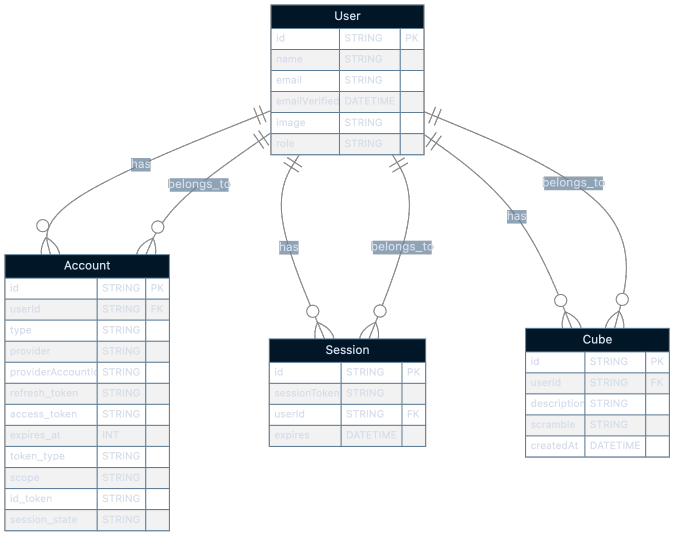

Figure 5.10: Entity-Relationship diagram of the database schema.

5.3.2 Prisma ORM

Prisma is used as the Object-Relational Mapping (ORM) tool for this project. Prisma's schema file (`schema.prisma`) defines the data models and their relationships in a declarative way. Prisma generates a type-safe client that can be used in the application code to perform database operations such as queries, inserts, updates, and deletes.

Some key benefits of using Prisma include:

- **Type Safety**: Prisma generates TypeScript types for all models and queries, reducing runtime errors and improving developer productivity.
- **Migrations**: Prisma provides a migration system that allows for safe and versioned changes to the database schema.
- **Performance**: Prisma optimizes queries and provides efficient data access patterns.
- **Developer Experience**: The Prisma Client API is intuitive and well-documented, making it easy to work with complex data models and relationships.

Overall, the combination of a well-structured relational schema and Prisma ORM ensures that the application can efficiently manage user data, authentication, and Rubik's Cube records with reliability and scalability.

5.4 Backend

The backend is split across two main parts: a Next.js app that powers most of the frontend and API logic, and a Python Flask service that handles the heavier, more algorithmic stuff (like solving the cube or analysing images). This setup keeps the interface responsive while offloading complex tasks to specialized services.

5.4.1 Next.js Backend

The Next.js App Router handles the REST API for everything from cube state management and solving steps to image processing and authentication. It keeps things fast, modular, and easy to plug into the frontend.

Authentication Endpoints

Authentication is handled using NextAuth, which makes it simple to plug in providers like Google and Facebook. It also deals with session handling so that each user stays securely connected.

NextAuth.js Routes

- **Endpoint:** `/api/auth/[...nextauth]`
- **Methods:** GET, POST
- **Purpose:** Handles everything related to signing in, signing out, session management, and OAuth callbacks
- **Features:**
  - JWT-based sessions
  - Prisma adapter for storing users
  - OAuth provider support
  - Custom sign-in routing
  - Automatic user creation

Cube Solving Endpoints

When the user clicks ”Fastest Solution”, the app sends the current cube state to a separate Flask service that runs Kociemba’s algorithm to generate the optimal solve sequence.

Kociemba Algorithm Service

- **Endpoint:** `GET /api/kociemba`
- **Params:** `cubeString` – The cube's state in Kociemba notation
- **What it does:** Forwards the request to Flask, which runs the two-phase Kociemba algorithm and returns a clean solution
- **External Service:** `${FLASK_URL}/kociemba`
- **Response:** The sequence of moves + move count
- **Use case:** Finding the shortest possible solution

Robot Control Endpoint

When the user clicks "Solve by Robot" or "Send to Robot", the app sends the computed solution sequence to a backend endpoint, which then communicates with the physical robot to execute the moves.

- **Endpoint:** `POST /api/robot`
- **Params:** `algorithm` – The solution sequence as a string of moves (e.g., `U1 D3 F2 ...`)
- **What it does:** Receives the solution from the frontend, formats it as needed, and forwards the move sequence to the robot controller (hardware or microservice) for execution.
- **External Service:** `${ROBOT_URL}/execute` (or similar, depending on deployment)
- **Response:** Status message indicating whether the robot accepted and started executing the moves (e.g., `200 OK, {"message": "Robot started"}`)
- **Use case:** Automating the physical solving of the cube by sending the optimal solution to the robot hardware.

Image Detection Endpoints

The app also lets users upload six face images of the cube to extract the current scramble. This logic is handled in Python using a computer vision pipeline.

Cube State Detection

- **Endpoint:** `POST /api/detection`
- **Content-Type:** `multipart/form-data`
- **Purpose:** Sends the six uploaded images to the Flask backend, which analyzes colours and determines the current state
- **External Service:** `${FLASK_URL}/detect`
- **Processing:**
  - Detects and classifies sticker colours
  - Extracts sticker positions for each face
  - Validates cube consistency
  - Applies confidence scoring and error correction
- **Response:** A 3×3 matrix per face representing the cube's state

Error Handling and Security

Security and stability are built into every endpoint. Here's how things are kept safe:

Authentication and Authorization

- JWT sessions are checked for every protected route
- Users can only manage their own cube state and actions
- Prisma and parametrized queries avoid SQL injection risks
- Cross-user access is blocked completely

Error Response Format

Consistent error messages help with debugging and clarity:

- **401 Unauthorized:** User isn't logged in or session is invalid
- **400 Bad Request:** Something's wrong with the input
- **404 Not Found:** Resource doesn't exist or isn't accessible
- **500 Server Error:** Something failed server-side

5.4.2 Flask Microservice for Detection and Solving

Alongside the Next.js API routes, a lightweight Python Flask microservice is responsible for handling the two heaviest operations in the app: cube state detection from images using OpenCV and solving the cube using Kociemba's algorithm. This service runs independently and is queried by the main Next.js backend when needed.

Endpoint: `POST /detect`

This endpoint receives up to six images—one for each face of the cube—and returns a 3×3 colour label matrix for each one.

- **Input:** Multipart form-data with a list of files under the `images` key.
- **Processing:**
  - Each image is converted to HSV format.
  - A 3×3 grid is extracted from the center of the image.
  - Each cell's colour is labeled using custom logic from the `helpers.py` module.
  - An annotated image is saved to disk for debugging and traceability (see Fig. 5.11).
- **Output:** For each image, the corresponding face grid is returned in the response, labeled by face colour (e.g., `red`, `green`, etc.).
- **Use Case:** Used when a user uploads cube face photos to auto-configure the digital cube.

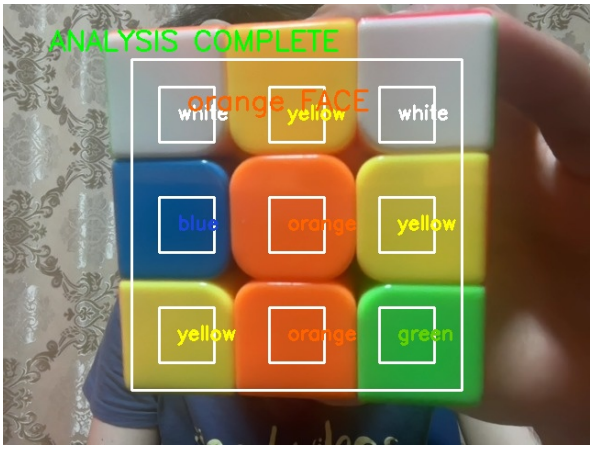

Figure 5.11: Analysis of an image performed by the backend.

Endpoint: `GET /kociemba`

This endpoint proxies cube-solving requests using a compact string representation of the cube. To compute optimal solutions for scrambled cubes, the Flask service uses the `RubiksCube-TwophaseSolver` library developed by Herbert Kociemba [Koc22]. This solver is widely known for efficiently solving the Rubik's Cube in at most 20 moves using a two-phase approach.

Library Features:

- Written in Python, ported from the original Java implementation
- Computes near-optimal solutions with minimal search depth
- Can be imported as a module via `import twophase.solver as sv`
- Returns a space-separated list of moves (e.g., `F U R2 B' ...`) with total move count

Upon the first run of the solver, the library takes up to half an hour (depending on the computational power of the device) to compute the prunning tables that are required for computing the solution on a fast manner. These tables take up to 80 MB of storage, and are used for all the future calls made to the endpoint.

The endpoint has a query parameter `cubeString`, which is a string consisting of 54 characters, representing the Kociemba representation of a scrambled cube. The letters stand for the six faces of the cube: **U** (Up), **L** (Left), **F** (Front), **R** (Right), **B** (Back), and **D** (Down).

The positions of the facelets are defined in Fig. 5.12:

```
                |************|
                |*U1**U2**U3*|
                |************|
                |*U4**U5**U6*|
                |************|
                |*U7**U8**U9*|
                |************|
************|************|************|************
*L1**L2**L3*|*F1**F2**F3*|*R1**R2**R3*|*B1**B2**B3*
************|************|************|************
*L4**L5**L6*|*F4**F5**F6*|*R4**R5**R6*|*B4**B5**B6*
************|************|************|************
*L7**L8**L9*|*F7**F8**F9*|*R7**R8**R9*|*B7**B8**B9*
************|************|************|************
                |************|
                |*D1**D2**D3*|
                |************|
                |*D4**D5**D6*|
                |************|
                |*D7**D8**D9*|
                |************|
```

Figure 5.12: Facelet positions of the Rubik's Cube using standard notation.

Move Notation

The library returns a single string, consisting of a sequence of moves required for solving the cube. The string is computed by several groups of the following form:

- A single letter corresponding to each one of the faces.
- Either 1, 2, or 3, indicating the number of times the move is repeated. Equivalently, 3 corresponds to the counter-clockwise rotation of the move.
- A single space.

At the end of the string, there is, written in paranthesis, the number of groups present in the string. In Kociemba representation, a group is considered only one move.

Chapter 6

Hardware Experiment: Robot

An interactive application of the computational solutions derived from Kociemba's Algorithm is realised through its integration into physical robots designed to solve the Rubik's Cube. These robots typically rely on an internal processing system that first analyses the scrambled configuration of the cube, and then computes an optimal or near-optimal solution using the algorithm. The mechanical structure of such robots is often custom-built, leveraging 3D-printed components to ensure precision and adaptability. Common hardware setups include servo motors or stepper motors for rotation, microcontrollers such as Arduino or Raspberry Pi for control logic, and open-source software for executing the move sequence. This synergy between software and hardware allows the robot to perform rapid and accurate solves, demonstrating a seamless integration of algorithmic theory and robotic engineering.

6.1 Physical Components

6.1.1 Structure

Cube Used

While any cube has the potential to be part of a robot, the one that is described in this thesis uses a MoYu Weilong V2. It is a high-performance 3x3 speedcube known for its smooth turning, excellent corner cutting, and lightweight design. Popular before magnetic cubes, it offered customizable tensioning and stability, making it a top choice for competitive speedcubers, and for a robot as well.

Figure 6.1: MoYu Weilong V2

Frame

The next important aspect of the robot is the frame. This has to offer robustness, while still being able to incorporate in it the Cube. The proposed design can be seen in Fig. 6.3. It was manually designed as a 3D object and printed on a 3D printer.

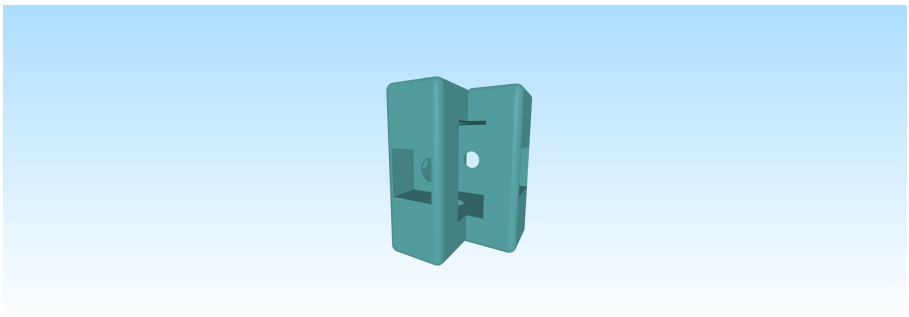

Figure 6.2: Frame of the robot

The frame is a robust, symmetrical cube-shaped structure featuring an open-face design that allows a Rubik's Cube to be enclosed at its center. Engineered for balance and precision, each side includes a dedicated space to mount a motor, enabling direct control of an individual cube face. The internal layout is carefully dimensioned to provide sufficient clearance for smooth rotation of all six faces. Manual assembly of the cube within the frame is required, ensuring a secure fit. Overall, the design combines mechanical strength with functional simplicity, making it ideal for automation or robotics applications.

Adapter

The final essential component of the design is the set of adapters. Each piece is specifically designed to connect a motor to a single face of the cube (with further details about the motor setup provided later in this chapter). The shape of each adapter is tailored to match the centre of a cube face, ensuring a snug and secure fit. One end of the adapter includes a dedicated slot that precisely accommodates the motor's axle. This ensures that every rotation of the motor is directly transferred to the corresponding face of the cube, enabling accurate and reliable movement as dictated by the program.

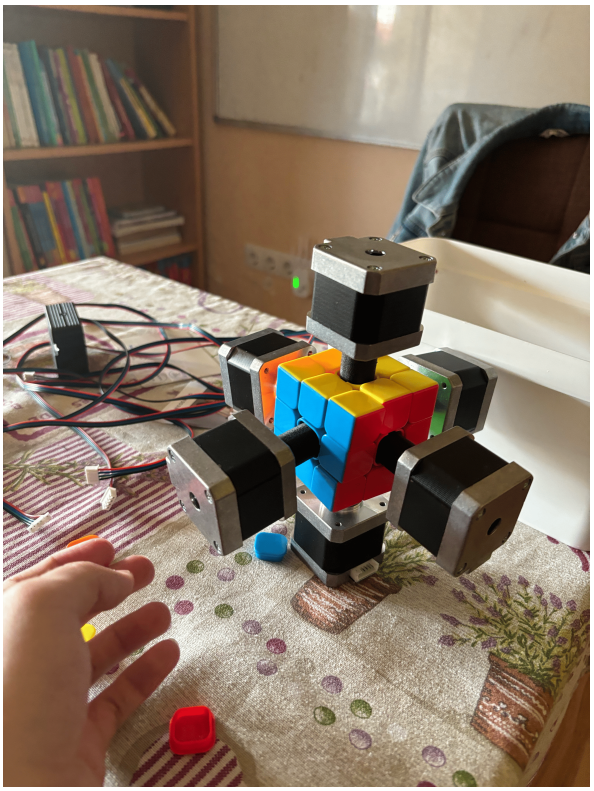

Figure 6.3: Motors with adapters attached to them

6.1.2 Hardware

Stepper Motors

The most important requirement of the robot is to be precise with its movements. Although not directly, the movement mechanism should also have enough force to rotate a face, as the main point of interaction of the motor is the centre of the face. A popular and reliable solution that satisfies both of these requirements is stepper motors. A stepper motor is a type of brushless DC electric motor that divides a full rotation into a number of equal steps. These motors convert electrical pulses into

discrete mechanical movements, allowing precise positioning and speed control. Each pulse sent to the motor causes it to rotate a precise angle, making them ideal for applications requiring accurate movement and positioning.

The system consists of 6 Nema17 stepper motors. A technical sketch of the motors can be seen in Fig. 6.4

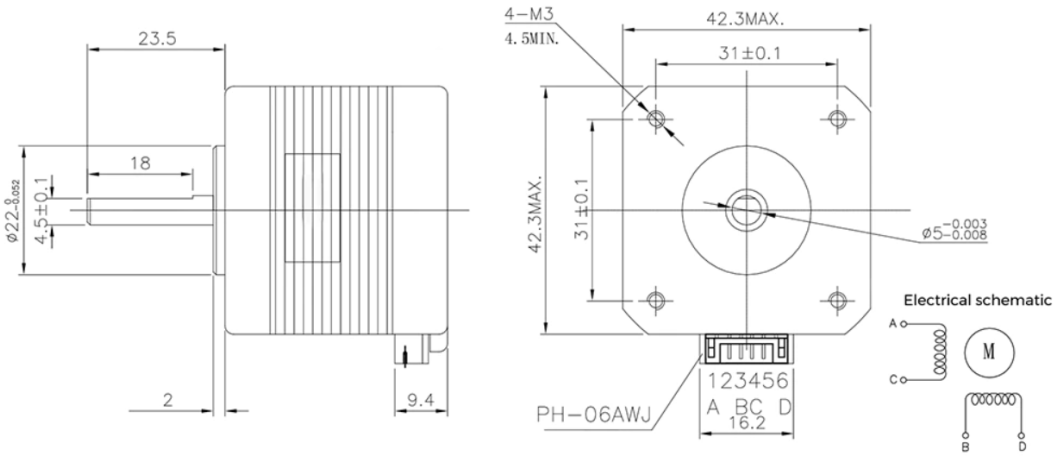

Figure 6.4: Technical drawing of the stepper motor

Raspberry PI

To control the precise movements of the six stepper motors, a central unit is required to process input commands and send accurate electrical signals. A popular and reliable choice for this role is the Raspberry Pi — a compact, affordable single-board computer. Its versatility and GPIO (General-Purpose Input/Output) capabilities make it ideal for hardware interfacing. In this setup, the Raspberry Pi acts as the brain of the robot, coordinating each motor's actions based on the programmed algorithm. Through software libraries and motor drivers, it ensures that each face of the cube is rotated at the correct moment and in the right direction, enabling the robot to execute complex move sequences reliably. [CBE25]

DRV8825 Motor Drivers

To bridge the control signals from the Raspberry Pi to the stepper motors, the system uses DRV8825 motor driver modules. These drivers are capable of handling higher current levels and provide microstepping support, which enables smoother and more precise motor movements. Each DRV8825 module receives pulse and direction signals from the Raspberry Pi's GPIO pins and translates them into appropriate current control for the Nema17 motors. Additionally, the drivers include built-in protection features such as over-temperature and over-current safeguards, making them a reliable choice for prolonged and repetitive operation. The use of

these drivers ensures that each face rotation is both accurate and consistent, crucial for the correct execution of cube-solving algorithms.

Wiring

Pin	Function
VMOT	Supplies power to the stepper motor (typically 8.2V–45V). Should be connected to an external power source suitable for the motor.
GND	Common ground for both motor power and logic control. Must be shared with the Raspberry Pi ground.
STEP	Receives pulse signals; each rising edge causes the motor to move one step. Connected to a GPIO pin on the Raspberry Pi.
DIR	Controls the direction of rotation. HIGH or LOW signal determines clockwise or counter-clockwise movement.
SLEEP	When pulled LOW, the driver enters low-power sleep mode. Typically tied HIGH for normal operation.
RESET	Resets internal driver logic. Often connected to SLEEP and kept HIGH in regular usage.
ENABLE	Enables or disables the motor outputs. Active LOW (pull LOW to enable). Useful for turning motors off when idle.
A1, A2, B1, B2	These pins connect to the two coils of the stepper motor. A1–A2 and B1–B2 form the two motor phases.

Table 6.1: DRV8825 Key Pin Descriptions

With all the components defined, the most important step is to ensure proper wiring. A stepper motor typically requires more current than the Raspberry Pi can supply. While the Raspberry Pi can generate the necessary control signals through its GPIO pins, it is not designed to power motors directly. Driving six stepper motors simultaneously would easily exceed the board's power limits. To provide sufficient energy, an external 12V power supply is used to deliver the required current to the DRV8825 motor drivers. This setup ensures stable and consistent operation without overloading the Raspberry Pi. There is the risk to fry the motors, so an electrical condenser is used as a barrier for the system.

The key pins listed in Table 6.1 play a vital role in the coordination between the Raspberry Pi and the stepper motors. In particular, the STEP and DIR pins must be

connected to the correct GPIO pins to ensure accurate pulse signaling and direction control. The `ENABLE` pin can be used to toggle motor activity, while `VMOT` and `GND` must be correctly powered and grounded to avoid instability. Proper connection of motor coil pairs to the designated output pins (`A1/A2`, `B1/B2`) is also essential for consistent movement. Ensuring that all these connections are stable and well-routed is critical for safe and predictable robot operation. A wiring schema can be seen in Fig. 6.5

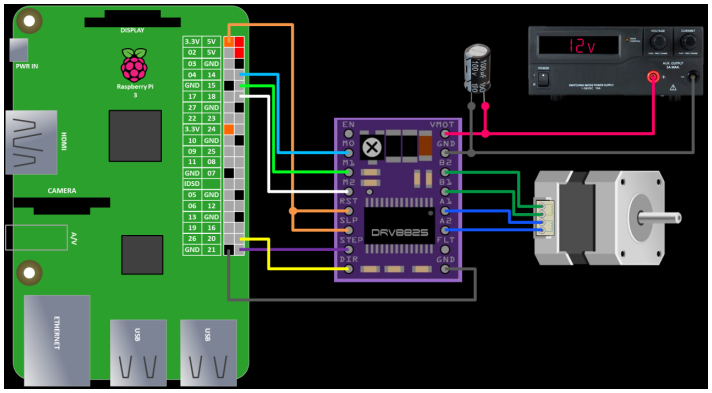

Figure 6.5: Wiring for one motor

The setup supports six motors using a breadboard to organize the DRV8825 drivers. All drivers are powered by a shared 12V supply connected to the breadboard. A ground wire links the power supply and Raspberry Pi to ensure a common reference for reliable signal transmission. The final assembly of the robot can be seen in Fig. 6.6

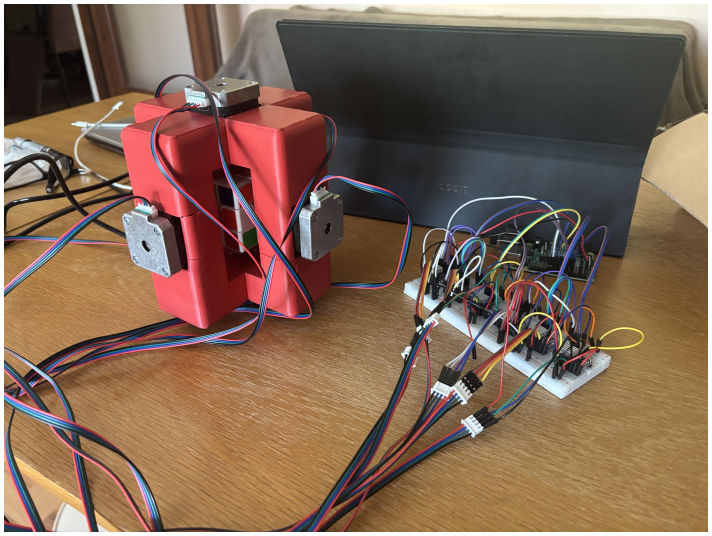

Figure 6.6: Robot with the Wiring complete

6.2 Software

As it relies on external input, the entire program is implemented as a server in Python using Flask. On the Raspberry Pi, this server runs continuously and listens for requests from other services. To make it accessible over the internet, the server is exposed using ngrok. Ngrok is a secure tunneling service that creates a public URL for a local server, allowing remote devices or applications to interact with the Raspberry Pi without complex network configurations or port forwarding. [Shr25]

Endpoint: `POST /move`

This endpoint receives a string with all the moves that should be made by the robot, to bring the cube to a desired state.

- **Input:** A string of an algorithm using Kociemba's representation
- **Processing:** Each move from the string is parsed, and based on it, the robot moves the corresponding face in the specified direction
- **Use Case:** Used to move the motors of the robot

This is the only endpoint required for the robot to interact with the exterior environment. This endpoint is used as well to scramble the cube.

6.3 Use case

There is a strong connection between the frontend application and the robot. When a user inputs a scrambled cube, the application captures its current state and sends it to the robot. To replicate the same configuration inside the robot, the program first computes the optimal solution using Kociemba's algorithm. It then reverses this sequence of moves, generating a new string that, when applied to a solved cube inside the robot, recreates the scrambled state provided by the user.

Chapter 7

Conclusions

This thesis presented the design and development of a software-hardware system capable of recognizing and solving a Rubik's Cube using visual and algorithmic methods. The main objective was to build an integrated application that captures the cube's configuration, computes an optimal solution via Kociemba's algorithm, and physically executes it through a Raspberry Pi-controlled robot.

Throughout the project, multiple domains were addressed—image processing, HSV-based color detection, optimal path-solving algorithms, stepper motor control, and robotic design. A Python (Flask) backend was developed and linked with an interactive web interface, while the mechanical platform was fully custom-built and 3D-printed.

Key accomplishments include:

- Visual recognition of cube faces from camera input;
- Optimal solution generation using the Kociemba algorithm;
- Physical execution through a custom robotic arm system;
- Seamless communication between frontend and robot over the network.

Current limitations lie in the robustness of color detection and the mechanical precision of the robot. However, the system is scalable and can be enhanced with deep learning-based recognition models or more advanced mechanical components.

This project proves the viability of a full-stack, interdisciplinary solution and provides a strong foundation for future work in robotics, computer vision, and autonomous systems.

7.1 Future Improvements

As described in Chapter 2, there are multiple algorithms available that provide viable methods for solving a Rubik's Cube. These algorithms can be integrated into

the application, giving users the option to switch between different solving strategies such as CFOP, Petrus, Roux, or ZZ.

Beyond algorithm variety, several directions could enhance the robustness and functionality of the system:

- **Machine Learning-Based Color Detection:** Replacing traditional HSV thresholding with a trained model (e.g., using CNNs) could improve face recognition accuracy under varying lighting conditions.
- **Adaptive Difficulty Mode:** For educational use, the system could include slower, step-by-step animations or explanations for beginners learning how to solve the cube.
- **Mechanical Precision Upgrades:** Improved motor calibration, stronger adapters, or use of magnetic encoders could enhance the precision and speed of physical execution.
- **Camera Integration:** A fixed camera system could automate the scanning process fully, removing the need for manual image upload or alignment.
- **Mobile App Integration:** A dedicated mobile app could allow users to scan the cube and control the robot wirelessly through Bluetooth or Wi-Fi.
- **State Verification System:** After execution, a secondary vision check could confirm that each move was correctly applied, offering feedback and error correction.
- **Battery-Powered Mode:** Making the robot wireless and portable by integrating a battery system would make it more practical for demonstrations or learning environments.
- **3D Simulation View:** A WebGL-powered real-time 3D simulation of the robot's actions could be shown to the user alongside the physical movement.

These improvements would not only increase the performance and flexibility of the system but also make it more accessible, educational, and resilient in real-world environments.

Bibliography

[CBE25] Eben Upton CBE. Raspberry pi, 2025. Accessed: 2025-06-14.

[Col25] Iain Collins. Next-auth, 2025. Accessed: 2025-06-14.

[Cub24] GAN Cube. Gan robot — smart rubik's cube solver, 2024. Accessed: 2025-06-14.

[CYCT10] Jun-Dong Chang, Shyr-Shen Yu, Hong-Hao Chen, and Chwei-Shyong Tsai. Hsv-based color texture image classification using wavelet transform and motif patterns. *Journal of Computers*, 20, 01 2010.

[Koc22] Kociemba. Rubiktwophase solver, 2022. Accessed: 2025-06-14.

[mrd25] mrdoob. Three js, 2025. Accessed: 2025-06-14.

[Pal10] Pallets. Flask, 2010. Accessed: 2025-06-14.

[Poi25a] Poimandres. Drei, 2025. Accessed: 2025-06-14.

[Poi25b] Poimandres. React three fiber, 2025. Accessed: 2025-06-14.

[Pri25] Prisma. Prisma, 2025. Accessed: 2025-06-14.

[Rec17] Guinness World Records. Robot breaks world record solving rubik's cube in 0.637 seconds, 2017. Accessed: 2025-06-14.

[Rec25] Guinness World Records. Fastest robot to solve a rubik's cube, 2025. Accessed: 2025-06-14.

[RKDD14] Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. The diameter of the rubik's cube group is twenty. volume 56, pages 645–670, 2014.

[Rom17a] Speedcubing Romania. Metoda fridrih, 2017. Accessed: 2025-06-14.

[Rom17b] Speedcubing Romania. Rezolvarea cubului rubik, 2017. Accessed: 2025-06-14.

[Sch81] Jaap Scherphuis. Thistlethwaite's algorithm, 1981. Accessed: 2025-06-09.

[Shr25] Alan Shreve. ngrok, 2025. Accessed: 2025-06-14.

[Spe24a] Speedcubing. Petrus method for solving the rubik's cube, 2024. Accessed: 2025-06-14.

[Spe24b] Speedcubing. Roux method for solving the rubik's cube, 2024. Accessed: 2025-06-14.

[Spe24c] Speedcubing. Zz method for solving the rubik's cube, 2024. Accessed: 2025-06-14.

[Ver25] Vercel. Nextjs, 2025. Accessed: 2025-06-14.