CAN THO UNIVERSITY

COLLEGE OF INFORMATION TECHNOLOGY AND COMMUNICATION

UTILIZATION OF COLOR DETECTION IN RUBIK'S CUBE SOLVING USING THE KOCIEMBA ALGORITHM

Course: Project on Basic Topics in Software Engineering (Project - Basic Topics)

Course Class: CT239H

Supervisor: Nguyen Thanh Khoa, Ph.D

Author: Vuong Phan Quoc Cuong

ID: B2203544

SEMESTER ..., 20... - 20...

Instructor Evaluation

Acknowledgements

I wish to express my sincere appreciation to Dr. Nguyen Thanh Khoa for his dedicated guidance and valuable insights throughout this research. I also extend my deep gratitude to my friends for their thoughtful advice and support, which greatly assisted me in completing this work. This project represents my earnest efforts, and I present it with heartfelt thanks.

Can Tho, 31/3/2025
Author
Vuong Phan Quoc Cuong

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

I PROBLEM DESCRIPTION

The Rubik's Cube is a 3D puzzle with six faces, each initially having a uniform color. The goal is to return the cube to its initial state, where each face contains only one color, by performing a series of rotations.

II THE GOAL TO ACHIEVE

This project aims to develop a demo application that scans a Rubik's Cube using a camera, recognizes the colors on each face, and simulates the solution using the Kociemba algorithm.

III SOLUTION APPROACH AND IMPLEMENTATION PLAN

The project will be implemented in several stages:

1. Capturing images with a webcam.
2. Processing all images to detect and classify colors with OpenCV.
3. Storing the detected colors in a structured format.
4. Simulating the Kociemba algorithm to understand its step-by-step process while by leveraging an existing library.

Chapter 2

THEORETICAL BACKGROUND

This chapter presents several key concepts and essential techniques for developing and refining the Rubik recognition and solution process.

I BASIC CONCEPTS

I.1 Rubik's Cube Basics

To effectively detect and solve the Rubik's Cube, it is important to understand its structure and standard notation. [3]

- **Faces**: The Rubik's Cube has six faces, each represented by a letter:
  - **U (Up):** The top face
  - **D (Down):** The bottom face
  - **L (Left):** The left face
  - **R (Right):** The right face
  - **F (Front):** The front face
  - **B (Back):** The back face
- **Clockwise 90°:** U, D, L, R, F, B

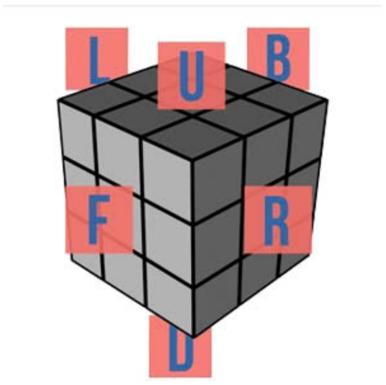


Figure 2.1: *Rubik's Cube Faces*

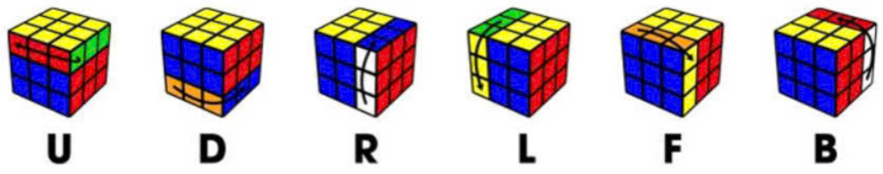


Figure 2.2: *Clockwise 90° rotations*

- **Counterclockwise 90° (denoted by a prime '):** U', D', L', R', F', B'

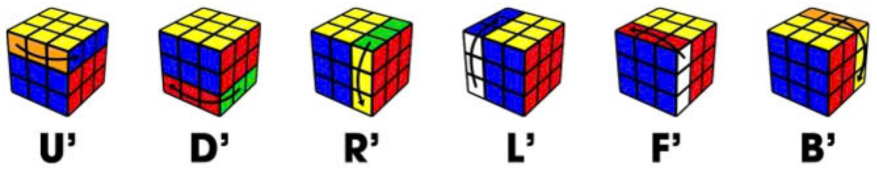


Figure 2.3: *Counterclockwise 90° rotations*

- **180° rotations (denoted by '2'):** U2, D2, L2, R2, F2, B2

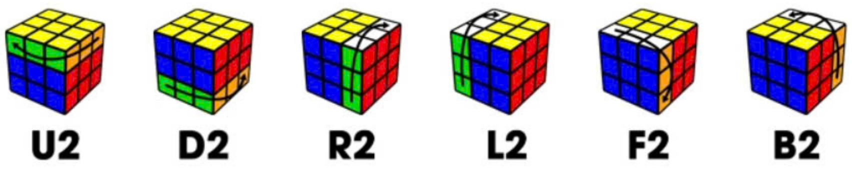


Figure 2.4: *180° rotations*

I.2 HSV Color Model

HSV (Hue, Saturation, Value) is a color model that represents colors based on three components:

- **Hue (H):** The shade or tint of the color, measured in degrees from 0 to 360.
- **Saturation (S):** The intensity or purity of the color, ranging from 0% to 100%.
- **Value (V):** The brightness or lightness of the color, ranging from 0% to 100%.

Essentially, the HSV model offers a straightforward approach to representing colors based on their hue, saturation, and value.

I.3 Kociemba's Algorithm

The Kociemba's Algorithm [2] is an optimal two-phase algorithm to solve a Rubik's Cube, developed by Herbert Kociemba in 1992. It works by reducing the cube to a specific subgroup in Phase 1, and then completely solving it in Phase 2. This technique usually produces shorter solutions **(fewer than 21 moves)** than methods based on solving layers.

II APPLYING THEORETICAL CONCEPTS

This section explores key aspects including state representation, color detection, rotation simulation, and solution generation, which together form an efficient and practical implementation.

II.1 Rubik's Cube State Representation

An ordinary Rubik's Cube (3x3) has six faces, each containing 9 squares. The state of the Rubik's Cube is stored in a list, where each element represents a square with a corresponding color. Each face of the Rubik's Cube is represented as follows:

$$S = [S_U, S_D, S_L, S_R, S_F, S_B] \tag{2.1}$$

Where:

- $S_U, S_D, S_L, S_R, S_F, S_B$: Lists containing 9 elements each, representing the six faces of the Rubik's Cube (Up, Down, Left, Right, Front, Back).
- Each element in the list represents a color (encoded as RGB values or symbolic values such as 'W' for white, 'R' for red, etc.).

Application in the Project: This structured representation allows the system to accurately track and store the state of the cube throughout the recognition and solving process.

II.2 OpenCV in Image Processing

OpenCV (Open Source Computer Vision Library) is an open source library widely used for image and video processing. It provides tools to detect objects, recognize colors, and perform various image manipulation tasks.

Application in the Project:

- OpenCV detects the colors of the Rubik's Cube faces using the HSV color model.

- Techniques such as color thresholding and contour detection convert the cube state into a flat six-face representation. This method accurately detects all six colors (white, red, blue, green, yellow, and orange) under varying lighting conditions, ensuring precise state extraction for the solver.

II.3 Rotation Logic and Notation

The standard Rubik's Cube moves are:

$$\{U, U', U2, D, D', D2, L, L', L2, R, R', R2, F, F', F2, B, B', B2\}$$

For explanations, see: Rubik's Cube Basics.

Application in the Project:

- The logic is implemented using Python if-else structures to simulate face rotations.

- Each rotation updates the color positions on the 2-D cube representation accordingly. The implemented logic ensures a precise simulation of all standard Rubik's Cube moves, validated through test cases.

II.4 Implementing Kociemba's Algorithm

Kociemba's two-phase algorithm efficiently finds the optimal solution for the Rubik's Cube.

Application in the Project:

- The Kociemba Python library is used to solve the Rubik's Cube based on the detected state.

- It generates concise rotation sequences, taking advantage of the algorithm's efficiency for practical implementation.

III PRACTICAL OUTCOMES

The following results demonstrate the successful application of theoretical concepts to the Rubik's Cube recognition and solving system:

- **Cube representation**: The state of the cube is accurately displayed in a flat 2D format, allowing seamless integration with the detection and solving processes.
- **Color detection**: OpenCV reliably detects all six colors (white, red, blue, green, yellow, orange) under varying lighting conditions, consistently recognizing the cube's state.
- **Logic of rotation**: The system precisely simulates standard Rubik's Cube moves (e.g., U, F, R2), validated through test cases with correct state updates.
- **Solving algorithm**: Kociemba's algorithm generates optimal solutions, averaging 20 moves or fewer, outperforming traditional layer-based methods in efficiency.

Chapter 3

Application Outcomes

This chapter presents the results obtained from developing and implementing the Rubik's Cube solver.

I PROBLEM ANALYSIS AND DATA STRUCTURES

To represent the Rubik's Cube in 2D, we adopt a cross layout for its six faces (`Up`, `Left`, `Front`, `Right`, `Back`, `Down`), arranged as follows:

```
    U
L   F   R   B
    D
```

I.1 Initializing the Coordinate Grid

The 2D interface relies on the `generate_grid(start_x, start_y, step_x, step_y)` function from the `utils` module. It generates a 3×3 grid of coordinates from a starting point and step sizes. Key grids are defined in `config.py`:

- `main`: Starts at (200, 120), step (100, 100) — for color scanning.
- `current`: Starts at (20, 20), step (34, 34) — real-time color display.
- `preview`: Starts at (20, 130), step (34, 34) — previous scan storage.
- Cube faces: Step (44, 44), e.g., `front` at (188, 280).

These grids support scanning, display, and cube visualization.

I.2 Rendering the 2D Interface

Using OpenCV, the application renders two main windows: `frame` for real-time scanning and color display, and `preview` for visualizing the cube's state.

- **Main Grid**: The `main` grid in the `frame` window is the scanning area where the Rubik's Cube face is placed to capture color data from the camera feed.
- **Current Grid**: The `current` grid, also in the `frame` window, displays the detected colors in real-time.

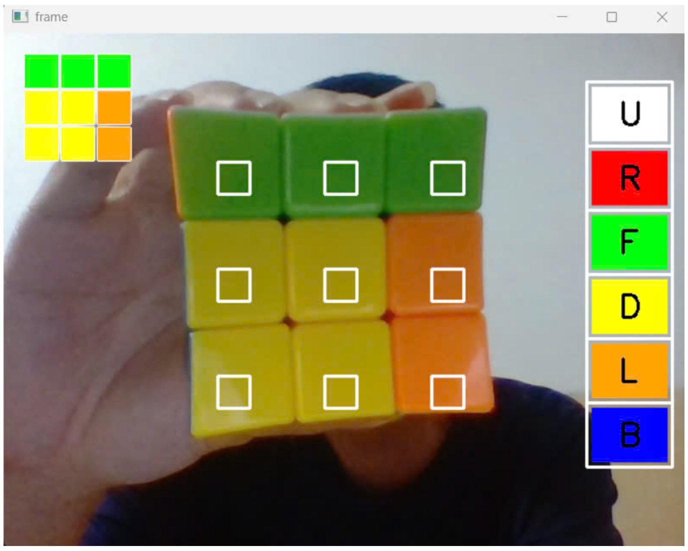


Figure 3.1: *Frame window showing main and current grids*

- **Preview Grid**: The `preview` window renders six 3×3 grids (`up`, `down`, `left`, `front`, `right`, `back`) to visualize the stored state of the cube, alongside interactive UI elements.

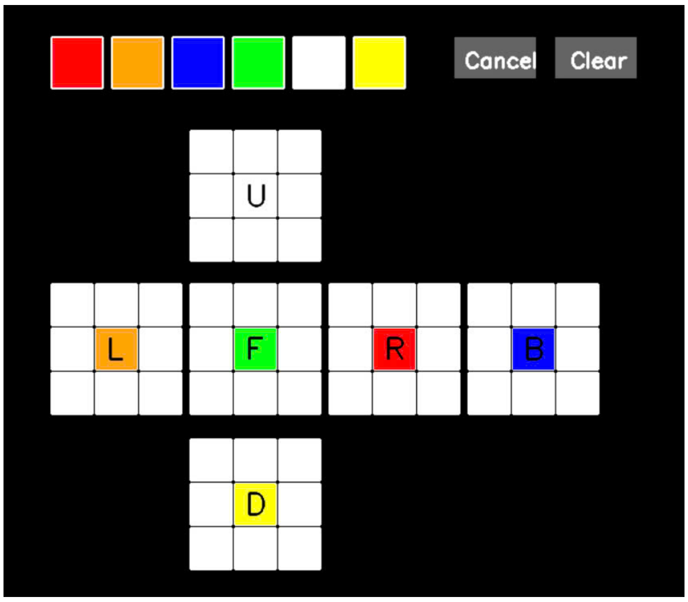


Figure 3.2: *Preview window with all cube faces*

II COLOR DETECTION USING HSV

[1] This section covers the process of detecting Rubik's Cube colors using the HSV color model, leveraging its ability to handle varying lighting conditions.

II.1 Determining HSV Values for Six Colors

To identify the six Rubik's Cube colors (red, orange, yellow, green, blue, white), specific HSV ranges are needed. A script in `HSV.py` uses a camera feed and OpenCV trackbars to find these ranges:

- Trackbars (`HMin`, `SMin`, `VMin`, `HMax`, `SMax`, `VMax`) adjust HSV bounds. Hue ranges from 0 to 179, Saturation and Value from 0 to 255.
- The camera captures a face, and trackbars are tuned until only the target color appears in the output.
- Changes in trackbar values print the min/max HSV ranges, e.g., `[49, 100, 66]`, `[179, 255, 255]` for red.

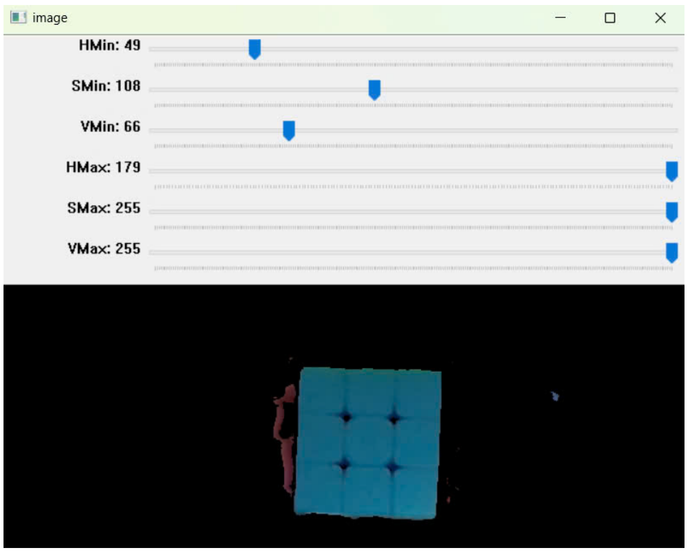


Figure 3.3: *Blue color detection in image processing*

II.2 Implementing the Color Detection Function

The `color_detect_kmeans()` function employs K-means clustering to classify colors based on HSV pixel values. It takes an array of HSV pixels and returns a list of corresponding color names. The implementation is as follows:

- **Data Normalization**: The input HSV pixels are normalized to ensure consistent clustering:
  - Hue (H) is divided by 180 to scale it to [0, 1].
  - Saturation (S) and Value (V) are divided by 255 to scale them to [0, 1].
- **K-means Clustering**: The normalized pixels are clustered using the K-means algorithm with 6 clusters (`n_clusters=6`), a fixed random seed (`random_state=0`), and 10 initializations (`n_init=10`) for stability.
- **Cluster Centroid Mapping**: The cluster centroids (returned in normalized form) are scaled back to their original HSV ranges (H $\times$ 180, S $\times$ 255, V $\times$ 255). Each centroid is then mapped to a color name based on predefined HSV ranges:

```
For each centroid (h, s, v):
    If (h < 10 or h > 170) and s >= 100 and v >= 50:
        Assign 'red'
    Else if 10 <= h <= 24 and s >= 100 and v >= 50:
        Assign 'orange'
    Else if 25 <= h <= 39 and s >= 100 and v >= 50:
        Assign 'yellow'
    Else if 40 <= h <= 84 and s >= 100 and v >= 50:
        Assign 'green'
    Else if 85 <= h <= 129 and s >= 70 and v >= 50:
        Assign 'blue'
    Else:
        Assign 'white' as the default
```

- **Label Assignment**: Each pixel is assigned the color name corresponding to its cluster's centroid based on the K-means labels.

In the main program (`main.py`), the `color_detect_kmeans()` function processes HSV values from the `main` grid:

```
For each position (x, y) with index i in the 'current' grid:
    Extract HSV values at index i
    Pass HSV values to color_detect_kmeans() to obtain color names
    Draw a 30x30 filled rectangle on the image at position (x, y)
    using the color matching the assigned color name
```

This approach leverages unsupervised learning to dynamically identify dominant colors, offering robustness over fixed range-based methods by adapting to the input data's distribution.

III ROTATION AND SOLVING ALGORITHMS

The algorithm design consists of two main components: rotation operations and the Kociemba algorithm.

III.1 Implementing Rotation Operations

Rotation operations turn a Rubik's Cube face either clockwise or counterclockwise, updating both the face itself and the edges touching it. These are handled in the `CubeState` class from `cube_state.py` with two functions: `rotate(side)` and `revrotate(side).`

Each function takes a `side` (like 'front' or 'up') and changes the cube's state:

- **Clockwise Rotation (`rotate`)**: Turns the chosen face 90° clockwise. For example, turning the front face can be described as:

```
Function rotate(side):
    If side is 'front':
        // Move edge stickers clockwise
        Set left[bottom] = down[bottom]
        Set up[bottom] = left[top]
        Set right[top] = up[bottom]
        Set down[bottom] = right[bottom]
        // Rotate front face 90° clockwise
        Swap front stickers: [0,1,2,3,4,5,6,7,8] -> [6,3,0,7,4,1,8,5,2]
```

- **Counterclockwise Rotation (`revrotate`)**: Turns the face 90° counterclockwise. The edge stickers move in the reverse direction, and the face shifts the opposite way. For the front face:

```
Function revrotate(side):
    If side is 'front':
        // Move edge stickers counterclockwise
        Set left[bottom] = up[bottom]
        Set up[bottom] = right[top]
        Set right[top] = down[bottom]
        Set down[bottom] = left[bottom]
        // Rotate front face 90° counterclockwise
        Swap front stickers: [0,1,2,3,4,5,6,7,8] -> [2,5,8,1,4,7,0,3,6]
```

These functions work for the six faces, making sure that the state of the cube stays correct after each turn, preparing for the application of Kociemba's second phase.

III.2 Comparison of CFOP and Kociemba Algorithms

The CFOP algorithm, also known as the Fridrich Method, is a popular speedcubing technique developed for solving the Rubik's Cube efficiently through a structured, layer-by-layer approach. Below is a comparison between the CFOP algorithm and the Kociemba algorithm, highlighting their differences in purpose, execution, and application.

Table 3.1: *Comparison of CFOP and Kociemba Algorithms*

Criterion	**CFOP**	**Kociemba**
Purpose	Optimized for human speedcubing with practice.	Designed for computers to find short solutions.
Steps	4 steps: Cross, F2L, OLL, PLL.	2 phases: G1 (oriented edges/corners), full solve.
Average Moves	55–60 moves, varies by skill.	18–20 moves, often suboptimal.
Solve Time	5–20 seconds (experts), over 1 minute (beginners).	0.1–1 second on modern PCs.
Complexity	Requires learning 119 algorithms, steep learning curve.	Computationally complex, fully automated.
Hardware	Only a Rubik's Cube and skill needed.	Needs CPU and memory for lookup tables.
Optimality	Non-optimal, prioritizes execution speed.	Near-optimal, fully optimal with more time.
Intuitiveness	Cross, F2L intuitive; OLL, PLL algorithmic.	Non-intuitive, based on group theory.
Competitive Use	Dominant in WCA, used by top speedcubers.	Unsuitable for humans, ideal for robots/software.

Note:

- F2L: First Two Layers.
- OLL: Orient Last Layer.
- PLL: Permute Last Layer.
- WCA: World Cube Association.
- G1: Group 1 state, where edges and corners are oriented.

III.3 Applying Kociemba's Algorithm

- Flowchart:

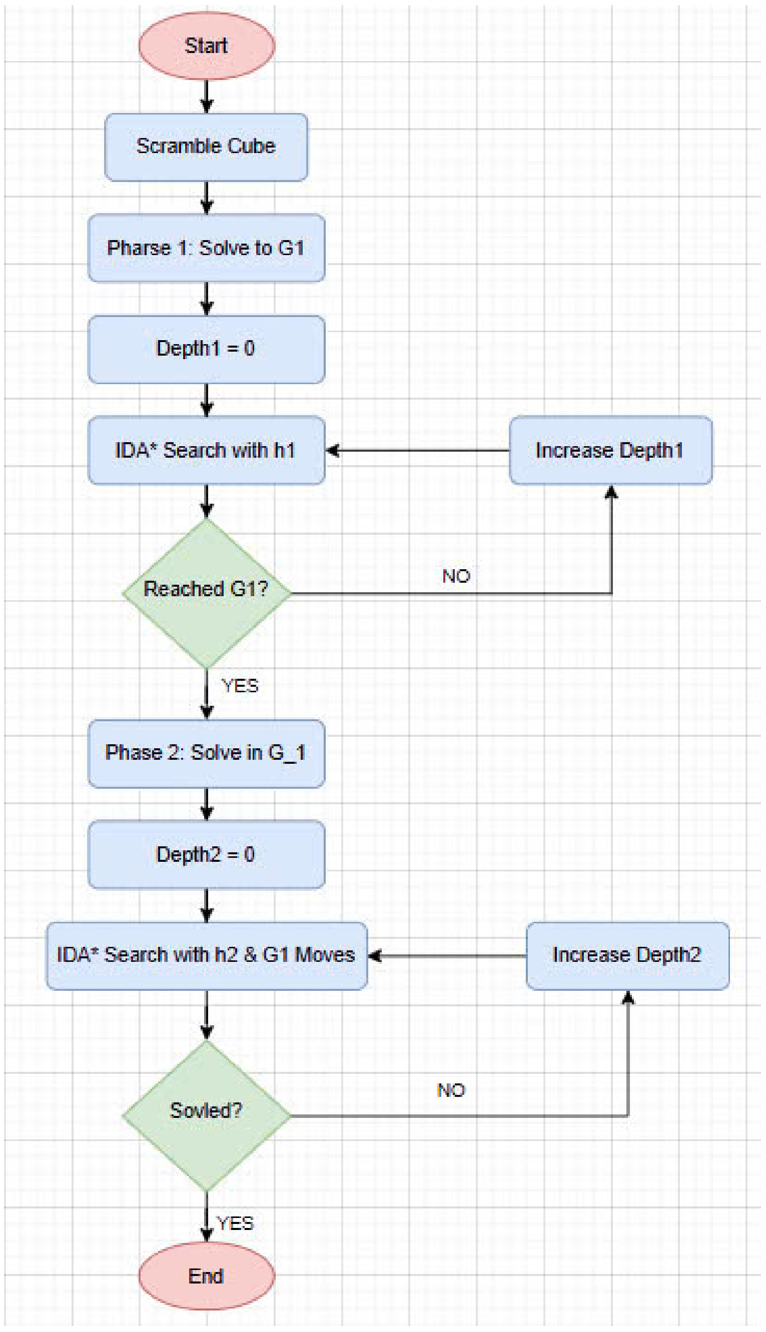


Figure 3.4: *Flowchart of Kociemba's Two-Phase Algorithm*

Kociemba's Two-Phase Algorithm offers an efficient approach to solving the Rubik's Cube by dividing the process into two phases, leveraging group theory and heuristic search techniques [2]. The cube's six faces are denoted U(p), D(own), R(ight), L(eft), F(ront), and B(ack), where U represents a 90-degree clockwise quarter turn, U' a 90-degree counter-clockwise turn, and U2 a 180-degree turn. A sequence of moves, such as U D R' D2, is termed a maneuver.

Phase 1: Transforming the Cube into Subgroup G_1

Phase 1 transforms a scrambled cube into the subgroup $G_1 = \langle U, D, R^2, L^2, F^2, B^2 \rangle$, where corner and edge orientations are fixed, and the four UD-slice edges (between U and D faces) are correctly positioned. The algorithm employs Iterative Deepening A* (IDA*) with a heuristic function $h_1(x, y, z)$, implemented as a lookup table, to estimate the minimum moves required to reach G_1, pruning branches up to 12 moves ahead.

The IDA* search incrementally tests maneuvers of increasing length until G_1 is reached. Pseudocode for Phase 1 is as follows:

```
function solvePhase1(cube):
    depth = 0
    while true:
        result = depthLimitedSearch(cube, depth, h1)
        if result.found:
            return result.maneuver
        depth = depth + 1

function depthLimitedSearch(cube, depth, h1):
    if isInG1(cube):
        return {found: true, maneuver: []}
    if depth <= 0 or h1(cube) > depth:
        return {found: false}
    for move in [U, U′, D, D′, R, R′, L, L′, F, F′, B, B′]:
        newCube = applyMove(cube, move)
        result = depthLimitedSearch(newCube, depth - 1, h1)
        if result.found:
            return {found: true, maneuver: [move] + result.maneuver}
    return {found: false}
```

Here, `isInG1` checks if orientations are correct and UD-slice edges are in place, while h_1 uses precomputed tables to ensure no overestimation.

- **Simulation Example:** Consider a cube scrambled by U R F' from a solved state (U: white, F: green, R: red). This misorients edge FL (green on F, white on L) and corner UBL (white on L). IDA* explores:
  - F adjusts edge FL to UF (white on U, green on F) and corner UBL to UFR (white on U).
  - R' further aligns edge FR to UR (white on U), reaching G_1 with maneuver F R' (2 moves).

Phase 2: Restoring the Cube within G_1

Phase 2 solves the cube within G_1 using only moves from {U, D, R2, L2, F2, B2}, permuting the eight corners, eight U/D-face edges, and four UD-slice edges. The heuristic $h_2(a,b,c)$ approximates the moves needed, guiding a similar IDA* search restricted to G_1 moves.

Pseudocode for Phase 2:

```
function solvePhase2(cube):
    depth = 0
    while true:
        result = depthLimitedSearchG1(cube, depth, h2)
        if result.found:
            return result.maneuver
        depth = depth + 1

function depthLimitedSearchG1(cube, depth, h2):
    if isSolved(cube):
        return {found: true, maneuver: []}
    if depth <= 0 or h2(cube) > depth:
        return {found: false}
    for move in [U, D, R2, L2, F2, B2]:
        newCube = applyMove(cube, move)
        result = depthLimitedSearchG1(newCube, depth - 1, h2)
        if result.found:
            return {found: true, maneuver: [move] + result.maneuver}
    return {found: false}
```

- **Simulation Continued:** From G_1 after F R', edges and corners are permuted but oriented. A single U' (1 move) aligns all pieces, yielding a total solution of F R' U' (3 moves).

Optimization

The algorithm iterates through suboptimal Phase 1 solutions (e.g., F R') to find shorter Phase 2 sequences (e.g., U'), stopping when Phase 2 requires zero moves, indicating optimality. For U R F', F R' U' is optimal, reversing the scramble directly.

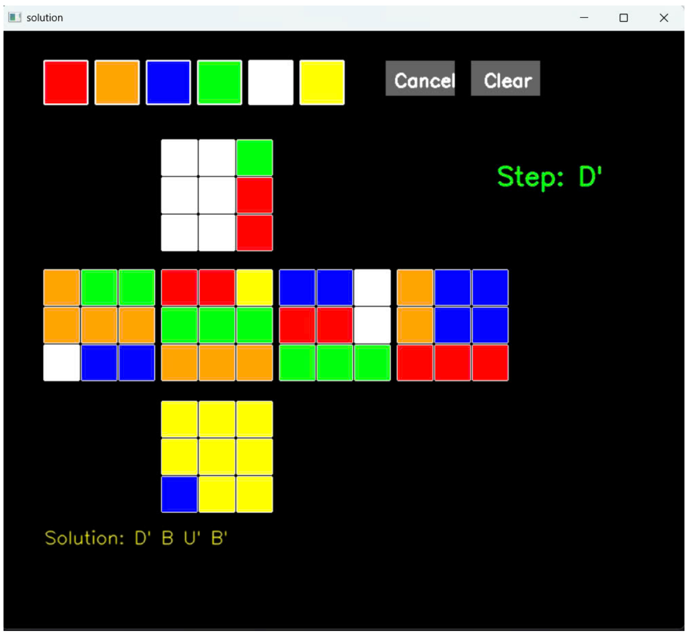
The implementation prioritizes speed over exhaustive optimality, avoiding maneuvers that re-enter Phase 2, though an Optimal Solver addresses this limitation [2].

IV PROGRAM INTRODUCTION

The Rubik's Cube solver features three windows:

- **Frame** for real-time color scanning (Fig. 3.1),
- **Preview** to store and visualize the cube's state (Fig. 3.2),
- **Solution** to display the solving moves (Fig. 3.5).

Colors are captured in Frame, assigned in Preview, and solved by pressing Enter to reveal the solution in Solution using Kociemba's algorithm.



Step: D'

Solution: D' B U' B'

Figure 3.5: *Solution window displaying solving moves*

Chapter 4

Conclusion and Evaluation

I RESULTS AND ACHIEVEMENTS

This project created a Rubik's Cube solver using Python and OpenCV, building a 2D model of the faces of the cube, detecting colors through the camera feed with the HSV model and solving it efficiently in under 20 moves using the Kociemba Two-Phase Algorithm.

II LESSONS LEARNED AND PROFESSIONAL EXPERIENCE

This project gave me practical experience with Python and OpenCV for computer vision, improved my grasp of HSV color detection in different lighting, and deepened my understanding of optimization and group theory via Kociemba's algorithm. It also sharpened my problem-solving and software development skills through debugging and real-time integration.

III ADVANTAGES

The solver provides a practical solution for Rubik's Cube enthusiasts, combining real-time color scanning with an efficient solving algorithm. Its use of Python and OpenCV ensures a lightweight, accessible implementation that runs on standard hardware. The clear separation of scanning (Frame), state management (Preview), and solving (Solution) enhances usability.

IV LIMITATIONS AND CAUSES

The solver faces several limitations:

- **Inaccurate color scanning**: Caused by low-quality cameras and fixed scanning positions, which prevent precise alignment and capture of the cube's colors.

- **Limited adaptability to lighting conditions**: caused by using hardcoded HSV thresholds implemented via conditional logic.
- **Slow operation**: The solver operates slowly because each face must be assigned manually.

V FUTURE DIRECTIONS

Future enhancements could include transitioning from a 2D to a 3D cube representation for a more intuitive visualization, replacing fixed HSV thresholds for centroid mapping with dynamic clustering methods to adaptively identify color boundaries based on input data, and training a machine learning model to improve color recognition accuracy under diverse lighting and camera setups.

Appendices

APPENDIX 1: DETAILED DEMO USAGE GUIDE

This section provides a step-by-step guide to using the Rubik's Cube solver demo:

1. **Setup**: Ensure Python and OpenCV are installed. Connect a webcam.
2. **Launch**: Run `main.py` to open the Frame window.
3. **Scan**: Place a cube face in the scanning area. Press 'u', 'd', 'l', 'r', 'f', or 'b' to assign colors to the respective face in Preview. Alternatively, if using the front-facing camera, you can press the on-screen buttons in the Frame window instead of the keyboard keys.
4. **Solve**: Once all faces are scanned correctly in Preview, press Enter to view the solution in the Solution window.

APPENDIX 2: FORMS, DOCUMENTS, AND FORMULAS

This section includes key formulas and configurations used:

- **HSV Color Detection**: Manually defined ranges (e.g., red: H: 0-10, S: 100-255, V: 100-255) using if-else logic in `color_detect()`.
- **Grid Coordinates**: Defined in `config.py`, e.g., `generate_grid(200, 120, 100, 100)` for the main scanning area.
- **Kociemba's Algorithm**: Two-phase approach for solutions under 20 moves.

APPENDIX 3: SOURCE CODE

The full source code is available at: `https://github.com/MPGranji/Rubik_Solver`. Key files include:

- `main.py`: Main program integrating scanning, state management, and solving.

- `cube_state.py`: Manages cube state and rotations.
- `ui.py`: Handles Frame and Preview window rendering.

Refer to the GitHub repository for detailed implementation and updates.

REFERENCES

[1] GeeksforGeeks. *Real-Time Object Color Detection using OpenCV*. 2025. URL: `https://www.geeksforgeeks.org/real-time-object-color-detection-using-opencv/` (visited on 03/20/2025).

[2] Herbert Kociemba. *Kociemba's Cube Solver*. 2025. URL: `https://kociemba.org/cube.htm` (visited on 03/19/2025).

[3] Rubik Online Vietnam. *Tong hop cac ki hieu rubik va quy uoc*. 2025. URL: `https://rubikonline.vn/tong-hop-cac-ki-hieu-rubik-va-quy-uoc` (visited on 02/22/2025).