Analyzing Time Efficiency and Computational Complexity of CFOP Algorithm in Rubik's Cube Solving

Ahmad Wicaksono - 13523121
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
ahmadwicaksono031004@gmail.com, 13523121@std.stei.itb.ac.id

***Abstract*— CFOP algorithm (Cross, F2L, OLL, PLL) remains one of the most widely utilized methods for solving the Rubik's Cube due to its balance between simplicity and efficiency. This study investigates the computational complexity and time efficiency of the CFOP algorithm, providing insights into its practical applications and limitations. A case study of Max Park, a world-renowned Rubik's Cube speed solver, exemplifies the algorithm's effectiveness. On June 11, 2023, Max Park set a world record by solving a standard 3x3 Rubik's Cube in an astonishing 3.13 seconds, showcasing the potential of CFOP for competitive solving. By breaking down the algorithm's steps and analyzing its performance under theoretical and real-world conditions, this research aims to bridge the gap between algorithmic complexity and practical efficiency. The findings offer valuable implications for both casual enthusiasts and developers of Rubik's Cube-solving algorithms.**

***Keywords*—Rubik's Cube, CFOP, Algorithm, Efficiency.**

I. Introduction

The Rubik's Cube, first introduced by Ernő Rubik in 1974, represents a compelling intersection of mathematics, spatial reasoning, and algorithmic design. With over 43,252,003,274,489,856,000 possible configurations in a standard 3×3×3 cube, solving this puzzle efficiently has long captured the attention of mathematicians, computer scientists, and puzzle enthusiasts. Among the many solving techniques developed. CFOP (Cross, F2L, OLL, PLL) algorithm is widely regarded as the most influential and systematically efficient method, particularly in the domain of competitive speed cubing. Its structured, stepwise approach combines algorithmic precision with human intuition, making it a subject of interest for both theoretical exploration and practical application.

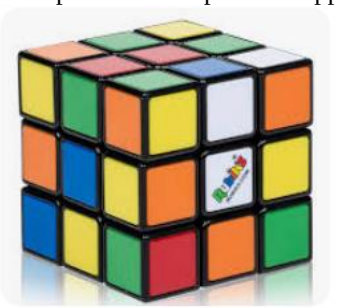

Figure 1.1 Rubik's Cube
(https://www.artofplay.com/products/rubiks-cube)

CFOP algorithm decomposes the problem of solving the Rubik's Cube into four sequential phases: constructing the cross (Cross), solving the first two layers simultaneously (F2L), orienting the last layer (OLL), and permuting the last layer (PLL). Each phase incorporates distinct algorithmic elements, often involving heuristic optimizations to reduce the total number of moves. Despite its wide adoption, the computational properties of CFOP remain underexplored, particularly in terms of its time efficiency and computational complexity across varying scenarios. A rigorous examination of these factors is essential for understanding their performance limits and identifying opportunities for optimization.

This research investigates the time efficiency and computational complexity of the CFOP algorithm, with a particular focus on its scalability and practical implications. The complexity of the algorithm arises from several factors, including the combinatorial nature of the cube's states, the branching factor in decision-making during the F2L phase, and the varying lengths and execution times of algorithms in the OLL and PLL phases. While CFOP is designed for efficiency in practice, theoretical analysis can shed light on its average-case and worst-case performance, as well as its dependence on human proficiency and algorithm memorization.

In summary, this research seeks to bridge the gap between practical solving strategies and theoretical computational analysis by systematically examining the CFOP algorithm. Through this investigation, the Rubik's Cube transcends its role as a recreational puzzle, serving as a model for exploring the interplay between algorithmic design, human intuition, and computational efficiency.

II. THEORETICAL BASIS

A. Time Complexity

Time complexity refers to a measure the amount of time an algorithm takes to execute. It is typically assessed by counting the number of basic operations an algorithm performs. Consequently, the time required by an algorithm is considered

proportional to the number of such operations, linked by a constant factor.

Since the execution time of an algorithm can vary for different inputs of the same size, the worst-case time complexity is often used. This represents the maximum time the algorithm might take for any input of a given size. Alternatively, the average-case time complexity, though less frequently analyzed, describes the average runtime over all possible inputs of a specific size. Both measures express time complexity as a function of the input size.

Exact computation of time complexity functions is often impractical, so the focus is usually on the algorithm's behavior as the input size becomes large—its asymptotic behavior. Time complexity is commonly represented using big O notation, such as $O(n)$, $O(nlogn)$, $O(n^2)$, or $O(2^n)$, where n represents the size of the input, often in bits.

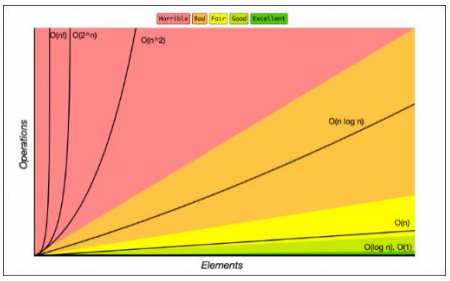

Figure 2.1. Time Complexity Chart (https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/)

Algorithms are categorized based on the function in the big O notation. For instance, an algorithm with $O(n)$ time complexity operates in linear time, while one with $O(n^{\alpha})$, where $\alpha > 0$ operates in polynomial time.

B. Rubik's Cube

Rubik's Cube invented in 1974 by Hungarian architect and professor named Ernő Rubik. It is created as a teaching tool to help students understand three-dimensional geometry, it quickly became a global phenomenon. The standard 3x3 Rubik's Cube consists of 26 smaller cubies arranged around a fixed core, with each of its six faces capable of independent rotation. The objective is to return the cube to its solved state, where each face displays a uniform color, after scrambling its pieces through a series of twists.

Beyond its appeal as a toy, the Rubik's Cube has deep roots in mathematics, particularly in group theory and combinatorics. The complexity of Rubik's cube has inspired the development of various solving methods, such as CFOP (Cross, F2L, OLL, PLL), Roux, and Petrus. Moreover, the Rubik's Cube has become a symbol of problem-solving and intellectual curiosity, influencing fields as diverse as artificial intelligence, optimization, and education.

As a cultural and mathematical artifact, the Rubik's Cube continues to inspire innovation. Speedcubing competitions have emerged, where participants compete to solve the cube in the shortest time, pushing human reflexes and algorithmic understanding to new limits. Meanwhile, researchers explore its theoretical implications, such as "**God's Number**" the maximum number of moves required to solve any scrambled cube.

To understand the mechanics of the Rubik's Cube, it is essential to familiarize oneself with the notation used to describe its moves. Each move corresponds to a rotation of one of the cube's six faces, and this notation provides a standardized language for solvers and algorithms alike. By breaking down these movements, one can analyze how the cube's pieces are repositioned and manipulated during the solving process. Below is an explanation of the standard moves, where each letter represents a specific face of the cube and the direction it is rotated.

- **R**: Rotate **right**-face clockwise.
- **U**: Rotate **upper** face clockwise.
- **F**: Rotate the **front** face clockwise.
- **D**: Rotate **down** face clockwise.
- **B**: Rotate the **back** face clockwise.
- **L**: Rotate **left** face clockwise.

Each of these moves can also be reversed into a **counterclockwise** move such as **R′**, **U′**, **F′**, **D′**, **B′**, **L′**, where the prime symbol (') indicates a counterclockwise rotation.

In addition to face rotations, whole-cube rotations are denoted by:

- **x**: Rotate the entire cube as if performing **R** while keeping the center fixed.
- **y**: Rotate the entire cube as if performing **U**.
- **z**: Rotate the entire cube as if performing **F**.

Middle layer moves involve rotating the internal slices of the cube:

- **M**: Rotate the middle slice parallel to the **L-R** axis.
- **E**: Rotate the middle slice parallel to the **U-D** axis.
- **S**: Rotate the middle slice parallel to the **F-B** axis.

`

C. CFOP Method

The **CFOP method,** also known as the **Fridrich method**, is one of the most used methods in speed solving a 3×3×3 Rubik's Cube. It is one of the fastest methods with the other most notable ones being Roux and ZZ. This method was first developed in the early 1980s, combining innovations by a number of speed-cubers. Jessica Fridrich, a Czech speed-cubers and the namesake of the method, is generally credited for popularizing it by publishing it online in 1997. Here are the 4 methods of CFOP:

1. Cross

This first stage of solving involves solving the four edge pieces around one center piece, matching the colors of that center and each of the centers of the adjacent sides, forming the eponymous cross shape on the first layer. Most beginner methods solve the cross by first putting the white edge pieces around the yellow center on the top, then matching them with the same colored center, and finally moving them down to match them with the white center. Most CFOP tutorials instead recommend solving the cross on the bottom side to avoid cube rotations and to get an overall better view of the important pieces needed for the next step (F2L).

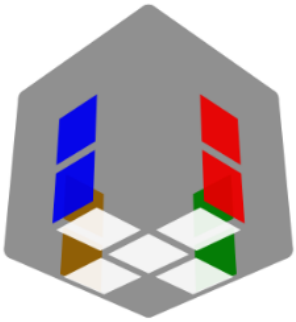

Figure 2.2. How Cross Solved in 3x3 Rubik's Cube (https://en.wikipedia.org/wiki/Rubik%27s_Cube)

This cross is usually solved intuitively, although some techniques, such as replacement, and edge orientation, are used. The white cross is most used for demonstration and by beginner and intermediate speedsolvers, though more advanced speedcubers can use any of the six colors to form the cross, a practice known as *color neutrality*. The cross can always be solved in 8 moves or fewer.

2. First Two Layer (F2L)

While the beginner methods continues by solving the four corners of the first layer and then matching the vertical edges to the corners to solve the second layer, the CFOP method solves each corner along with its vertical edge at the same time. There are 42 unique cases for the permutations of a corner and its matching edge on the cube (one of which corresponds to the solved pair), and the most efficient algorithm to solve any other case without "breaking" any already-solved pair is known and can be memorized.

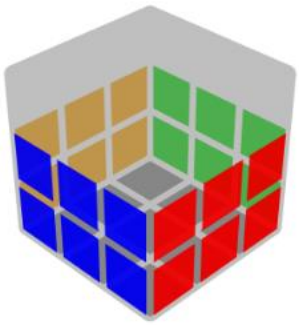

Figure 2.3. How F2L Solved in 3x3 Rubik's Cube (https://en.wikipedia.org/wiki/Rubik%27s_Cube)

3. Orient Last Layer (OLL)

This stage involves manipulating the top layer (yellow, if the cross is solved on white) so that all the pieces have the correct color on top, while largely ignoring the sides of these pieces. Doing this in one step is called "Full OLL". There are 58 possible combinations of piece orientations, so once again ignoring the solved case, this stage involves learning a total of 57 algorithms.

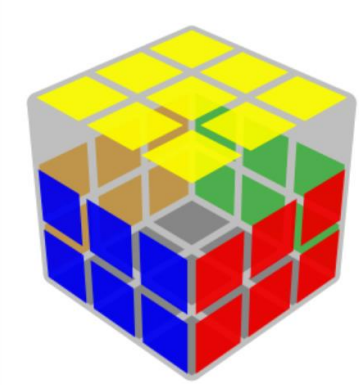

Figure 2.4. How PPL Solved in 3x3 Rubik's Cube (https://en.wikipedia.org/wiki/Rubik%27s_Cube)

4. PLL

The final stage involves moving the pieces of the top layer while preserving their orientation. There are a total of 21 algorithms for this stage. They are distinguished by letter names, often based on what they look like with arrows representing what pieces are swapped around such as, A-perm, F-perm, T-perm etc. Two-look PLL solves the corners first, followed by the edges, and requires learning just six algorithms of the full PLL set. The most common subset uses the A-perm and E-perm to solve corners (as these algorithms only permute the corners), then the U-perm (in clockwise and counter-clockwise variants), H-perm and Z-perm for edges. However, as corners are solved first in two-look, the relative position of edges is unimportant, and so algorithms that permute both corners and edges can be used to solve corners. The J, T, F, and R-perms are all valid substitutes for the A-perm, while the N, V and Y-perm can do the same job as the E-perm. Even fewer algorithms can be used to solve PLL.

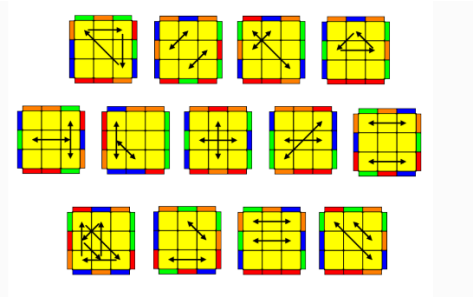

Figure 2.5. PLL Cases in 3x3 Rubik's Cube (https://en.wikipedia.org/wiki/Rubik%27s_Cube)

III. Algorithm Implementation

In this research, various algorithms for solving and analyzing Rubik's Cube configurations were implemented using the "PyCubing" library, which provides efficient tools for working with Rubik's Cube puzzles.

A. Cross Algorithm

The computational complexity of the cross algorithm is constrained by the finite number of states in a Rubik's Cube, its efficiency depends on the initial configuration. In the worst case, aligning each edge may require several rotations making the time complexity proportional to the number of misalignments. The current implementation uses efficient processes to minimize unnecessary moves. This function provides a solid foundation

for Rubik's Cube-solving research particularly in combinatorial optimization and process efficiency for adaptation to other solving frameworks.

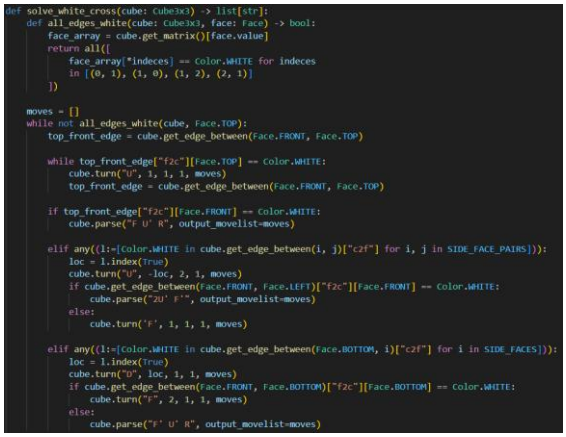

Figure 3.1. Cross Algorithm
(Writer's Archive)

"solve_white_cross" function is designed to solve the white cross on a 3x3 Rubik's Cube. This algorithm systematically aligns all white edges on the top face while ensuring these edges match their adjacent center colors. To achieve this, the function operates in two phases: first, gathering all white edges to the top face, and second, aligning and inserting them into their correct positions. In the initial phase, the algorithm iteratively checks the location of each white edge, employing predefined moves, such as F U' R, to reposition misaligned pieces. It handles complex scenarios, including edges on the sides or bottom face, using conditional logic and predefined face relationships. Once all edges are on the top face, this algorithm transits to the second phase, where it aligns the edges with their respective centers and inserts them into the solved state using efficient rotations.

B. Front Two Layers (F2L) Algorithm

Figure 3.2. First Layer Algorithm (F2L)
(Writer's Archive)

The provided functions such as solve_first_layer_corners and solve_second_layer_edges address two critical steps in solving a 3x3 Rubik's Cube by completing the first layer's corners and solving the second layer's edges. Both functions employ modular subroutines for checking the state of specific pieces, ensuring correct placement and orientation.

The solve_first_layer_corners function focuses on permuting and orienting white corners by identifying their positions and applying move sequences like "sexy moves" to either reposition or insert them. The solve_second_layer_edges function uses systematic checks to locate unsolved edges and applies efficient insertion algorithms to align and place them correctly.

The complexity of these functions arises from iterating over possible piece states with a worst-case scenario of handling multiple misaligned corners or edges. Each piece requires conditional checks and potentially multiple moves to reach the desired position resulting in an overall time complexity proportional to the number of misaligned pieces. These functions provide a foundational approach to solving the cube and offer opportunities for research into process-based optimization.

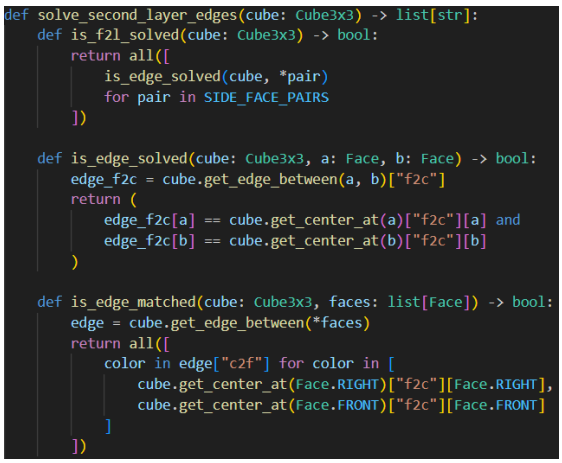

Figure 3.3. Second Layer Algorithm (F2L)
(Pycubing)

C. Orient Last Layer (OLL) Algorithm

The solve_oll_edges and solve_oll_corners functions address the Orientation of the Last Layer (OLL) step in solving a 3x3 Rubik's Cube by orienting the edges and corners of the top layer to align with the yellow face. The solve_oll_edges function handles three main patterns such as dot, line, and L-shape using specific algorithms to orient edges. Similarly, solve_oll_corners iteratively adjusts corner orientation using "sexy moves" and bottom-layer rotations with a bounded number of iterations due to the finite corner configuration and also resulting in a time complexity of $O(1)$.

Both functions are deterministic to ensure efficiency. While these implementations are optimized for manual solving, their performance could be further enhanced by integrating pattern recognition or advanced algorithms, making them suitable for exploring process efficiency and OLL algorithmic improvements in Rubik's Cube solving.

```python
def solve_oll_corners(cube: Cube3x3) -> list[str]:

    def is_oll_solved(cube: Cube3x3, face: Face) -> bool:
        return len({
            cube.get_corner_between(*pair, face)["f2c"][face]
            for pair in SIDE_FACE_PAIRS
        }) == 1

    moves = []
    cube.parse('x2', output_movelist=moves)
    while not is_oll_solved(cube, Face.BOTTOM):
        match cube.get_corner_between(Face.RIGHT, Face.FRONT, Face.BOTTOM)["c2f"][Color.YELLOW]:
            case Face.BOTTOM: sexy_moves = 0
            case Face.FRONT: sexy_moves = 4
            case Face.RIGHT: sexy_moves = 2
            case _: raise ImpossibleScrambleException("Cube could not be solved.")
        cube.parse(sexy_move_times(sexy_moves), output_movelist=moves)
        cube.turn("D", 1, 1, 1, moves)
    cube.parse('x2', output_movelist=moves)
    return moves
```

Figure 3.2. OLL Algorithm (solve_oll_corners) (Writer's Archive)

D. Permutation Last Layer (PLL) Algorithm

```python
def solve_pll_edges(cube: Cube3x3) -> list[str]:
    moves = []
    cube_matrix = cube.get_matrix()
    edge_swap = (sexy_move_times(1) + sexy_move_times(1, left_hand=True)
                 + sexy_move_times(-1) + sexy_move_times(-1, left_hand=True))

    for _ in range(2):
        match sum((1:=[
            cube_matrix[face.value][0, 1] != cube_matrix[face.value][0, 0]
            for face in SIDE_FACES
        ])):
            case 0: pass
            case 2:
                cube.parse(reverse_moves(moves))
            case 4: cube.parse(edge_swap, output_movelist=moves)
            case 3:
                loc = 1.index(False)
                cube.turn("U", -loc, 1, 1, moves)
                if (
                    cube_matrix[Face.LEFT.value][0, 1].value
                  - cube_matrix[Face.LEFT.value][0, 0].value
                ) % 4 == 2:
                    cube.parse(edge_swap, output_movelist=moves)
                else:
                    cube.parse(edge_swap * 2, output_movelist=moves)

    while np.unique(cube_matrix[Face.FRONT.value]).shape != (1,):
        cube.turn("U", 1, 1, 1, moves)

    return moves
```

Figure 3.2. PLL Algorithm (solve_pll_edges) (Writer's Archive)

The solve_pll_corners and solve_pll_edges functions address the Permutation of the Last Layer (PLL) step. The solve_pll_corners function utilizes the T-permutation algorithm to resolve diagonal or adjacent swaps in $O(1)$ moves due to the finite state space of corner permutations. Similarly, solve_pll_edges applies predefined edge-swap algorithms and checks for parity issues, iterating at most twice, with each iteration bounded by the limited edge configurations, leading to a time complexity of $O(1)$.

Both functions rely on efficient matrix-based checks and heuristic-driven swaps, ensuring deterministic and optimized performance for manual solving scenarios. However, the current design does not dynamically handle parity corrections, potentially requiring external intervention for some configurations, and could benefit from integrating more advanced algorithms or precomputed lookup tables for further optimization.

E. Edge Cases Handling

In NxN Rubik's Cubes, solving edge cases significantly impacts computational complexity, particularly during steps that involve pairing and positioning edge pieces. Unlike the standard $3x3$ cube, an NxN cube's edge consists of multiple smaller edge pieces (specifically $n-2$ pieces per edge). The requirement to correctly pair these smaller pieces introduces additional complexity as the cube's size increases.

The complexity arises from the iterative pairing process. For an NxN cube, there are $12(n-2)$ edge pieces, and solving these involves scanning, identifying, and aligning misplaced pieces. Each edge pairing step may involve multiple rotations, or slice moves to position edge fragments adjacent to their corresponding pieces. Since edge-solving generally requires evaluating the positions of all edge pieces, the algorithm operates over a search space proportional to n^2, resulting in a complexity of $O(n^2)$. This quadratic relationship is further compounded by potential edge parity issues unique to even-dimension cubes like 4x4 and 6x6, requiring additional algorithms to resolve misalignments.

Furthermore, as n increases, the higher number of intermediate states and possible configurations for edge piece placement exacerbates the computational effort. This complexity highlights the need for efficient pairing strategies to mitigate the performance impact in larger NxN cubes.

IV. ANALYSIS

The time efficiency and computational complexity of the Rubik's Cube solver code depend on several factors, primarily the cube's size (NxN) and the specific algorithms used for each solving phase. In the case of a general NxN solver (whether for 3x3, 4x4, or larger cubes), each algorithm must account for the increased number of pieces and potential misalignments as the cube size grows. For example, the "first layer corners" solving step involves multiple rotations to align corner pieces and position them correctly. As the cube's size increases, these rotations multiply, leading to a quadratic complexity of $\boldsymbol{O(n^2)}$. The "second layer edges" algorithm faces a similar challenge, where the complexity arises from the need to evaluate edge placements and swap or rotate edge pieces until they are correctly aligned.

For algorithms like "OLL (Orientation of the Last Layer)" and "PLL (Permutation of the Last Layer)", the time complexity is highly dependent on the number of pieces to be oriented or permuted. For larger cubes, more moves are required to reposition each edge or corner, and algorithms that were originally linear for the $3x3$ may become more intricate as the cube size increases. Given that the edge pieces in an NxN cube are represented by multiple smaller elements which must be correctly aligned or swapped, the overall complexity of solving the entire cube can grow nonlinearly with certain steps requiring handling edge parity and additional algorithms for fixing impossible configurations.

```
['2R2', 'F2', 'U2', 'D2', 'F2', '2Dw2', 'F2', 'U'
, "2Dw'", 'F2', 'U', "2Dw'", 'F2', 'U', "2Dw'", '
F2', '2Dw', 'U2', 'R', 'U', "R'", "U'", 'R', 'U',
 "R'", "U'", 'R', 'U', "R'", "U'", '2Dw', "U'", '
R', 'U', "R'", "U'", 'R', 'U', "R'", "U'", 'R', '
U', "R'", "U'", '2Dw', "U'", 'R', "U'", "R'", 'y'
, "L'", "U'", 'L', 'U', 'y', 'R', 'U', "R'", "U
", "2Dw'", 'R', 'U', "R'", "U'", 'y', "L'", "U'",
 'L', 'U', 'y', '2Dw2', "U'", 'y', "L'", "U'", '
L', 'U', 'y', 'R', 'U', "R'", "U'", '2Dw', 'U2',
 'y', "L'", "U'", 'L', 'U', 'y', 'R', 'U', "R'",
 "U'", '2Dw', 'U2', 'y', "L'", "U'", 'L', 'U', "y
'", 'R', 'U', "R'", "U'", '2Fw', 'R', 'U', "R'",
"U'", "2Fw'", 'x2', 'R', 'U', "R'", "U'", 'R', 'U
', "R'", "U'", 'D', 'U', 'R', "U'", "R'", 'U', 'R
', "U'", "R'", 'D', 'x2', 'U', 'R', 'U', "R'", "U
'", "R'", 'F', 'R2', "U'", "R'", "U'", 'R', 'U',
"R'", "F'", "U'", 'R', 'U', "R'", "U'", "L'", "U'
", 'L', 'U2', 'R', "U'", "R'", "U'", "L'", 'U', '
L', "U'"]
```

Figure 4.1. Example of Rubiks 3x3 Solved Moves
(Writer's Archive)

The solving counts for the $2x2$ and $3x3$ Rubik's cubes can be considered $O(1)$ complexity, as these cubes require a fixed number of steps that do not scale with their size. The $2x2$ cube, being a simplified version without edges, inherently has fewer permutations and can be solved in a consistent number of moves (95-100 Moves on average). Similarly, the $3x3$ cube can be efficiently solved using standard algorithms like CFOP within a finite move count (135-150 Moves on average). In contrast, higher-order cubes like the $4x4$ and $5x5$ introduce additional challenges, such as parity errors and edge pairing which makes their solving process scale more dramatically, different from the constant-time $O(1)$ complexity seen in the 2x2 and 3x3 cubes. Here is the data of the cube solved in a random scrambled:

Rubik's	Scramble	Solving Count (Moves)
2x2	[F U F U2 R2 F U' F' U	95
3x3	[B2 L B2 R U F2 L' B F U R' D2 U' R2 D L' F2 L2 D L R2 D' U' R' U]	139
4x4	[B L' D2' L B D2 B D' F U' D R2' B22 F' R2' U2 D2' R2 D2 U22 F2 R2' F2 U2' L2 R2 U2 B2 R' F' U2 R D2 F2 B2 D2 U2 B D2' L]	757
5x5	[R D' L22 D2 R' L' D2' L D22 L2' D22 U2 L2 R2' B2 F22 L2 B2 R22 F2' U2 R2 B2 R22 L2' U22 B' R D2 R22 D2 B' U' B2' L' D' L D' R' L22 U2 B22 R' L2 U2 B' D22 B' D F22 B22 U2 R22 B2 L22 D2 L' B2 U22 B2]	970

Figure 4.2. Data Counting Solving Moves on Rubik's Cube NxN
(Writer's Archive)

As the size of the Rubik's Cube increases from $6x6$ to $10x10$, the number of moves required to solve it grows significantly, following an increasing complexity pattern. This increase in moves reflects the added complexity of solving cubes with more pieces, additional edge and center alignment challenges, and the need to handle more complex parity errors. The growth in move count illustrates the **O(n²)** time complexity, where **n** represents the number of pieces on one side. This scaling shows how the larger state space of each cube size leads to more computational effort, as each new layer adds more pieces to manage, making larger cubes progressively more difficult to solve. Here is the given count solved moved in $6x6$ to $10x10$:

- $\boldsymbol{6x6}$ Rubik's cube solved within **1525** moves
- $\boldsymbol{7x7}$ Rubik's cube solved within **1998** moves
- $\boldsymbol{8x8}$ Rubik's cube solved within **2847** moves
- $\boldsymbol{9x9}$ Rubik's cube solved within **3376** moves
- $\boldsymbol{10x10}$ Rubik's cube solved within **4138** moves

The time complexity of solving Rubik's Cube puzzles increases with the size of the cube, particularly from the $2x2$ to $5x5$ cubes, and scales almost quadratically with larger cubes like the $6x6, 7x7$, and beyond. For smaller cubes, such as the $2x2$ or $3x3$, the solving process typically involves fewer moves, and the complexity grows **linearly** as more pieces are added. However, when moving to larger cubes, the problem becomes significantly more complex. For instance, a $6x6$ cube requires around 1525 moves to solve, a $7x7$ requires 1998 moves, and an $8x8$ requires 2847 moves. This growth is due to the increased number of pieces that need to be permuted and aligned, along with the introduction of additional complexities like center piece solving and edge pairing, which are not present in smaller cubes.

In larger cubes such as $9x9$ and $10x10$, the move count continues to increase, reaching 3376 and 4138 moves, respectively. This escalating number of moves results from several factors, such as the greater number of individual pieces (edges and corners) that must be correctly positioned and aligned. Additionally, larger cubes often introduce special parity issues that require unique algorithms to resolve, further increasing the number of necessary moves. For example, while a $3x3$ cube can typically be solved in fewer moves, solving a $5x5$ cube involves extra steps like solving the centers and pairing the edges, both of which increase complexity.

The relationship between cube size and solving moves demonstrates an approximately quadratic growth pattern, aligning with the time complexity of $O(n^2)$ where n is the dimension of the cube. The number of moves required grows in proportion to the square of the cube size, reflecting the increase in layers, pieces, and special cases that emerge as the cube dimensions grow. This trend suggests that the time complexity for larger cubes increases substantially as the solver must account for the additional layers, pieces, and complexities such as parity errors. While $O(n^2)$ is a suitable classification for complexity to reach the solved state.

V. CONCLUSION

In conclusion, the time efficiency and computational complexity of solving Rubik's Cubes can be analyzed with respect to the increasing size of the cube. For smaller cubes, such as the $2x2$ and $3x3$, the complexity is manageable, and human solvers can efficiently use methods like CFOP (Fridrich Method) with a time complexity that is generally considered $O(n\text{^}2)$. Ay the size of the cube increases, the number of pieces and configurations increases exponentially which increases the complexity and the number of moves required to solve the cube. For cubes larger than $5x5$, the $\boldsymbol{O(n^2)}$ complexity becomes more evident as the solver needs to handle additional pieces, manage edge pairing and address potential parity errors, which significantly increases the move count.

The $\boldsymbol{O(n^2)}$ complexity is primarily influenced by the need to solve multiple layers and align edges for each face of the cube, and this complexity grows as the cube's size increases. For instance, solving a $6x6$ cube can require upwards of 1500 moves, with the move count increasing further for larger cubes. While CFOP remains an efficient method for human solvers, it becomes less practical for larger cubes where more advanced

strategies are necessary. In the case of cubes larger than $5x5$, the solver's time efficiency is heavily influenced by how well they can adapt to the increasing complexity, and the $\boldsymbol{O(n^2)}$ growth in required moves becomes increasingly noticeable as the cube size grows. While CFOP is optimal for $2x2$ and $3x3$ cubes, it requires modifications and greater expertise for larger cubes, where time efficiency becomes more dependent on solving techniques suited to handling the increased number of configurations.

VI. Acknowledgment

The writer would like to express heartfelt gratitude to God for His continuous blessings, wisdom, and guidance throughout this academic journey. Without His divine assistance, the completion of this research paper would not have been possible.

The writer would also like to appreciate the Discrete Mathematic lecturers, Ir. Rila Mandala, M.Eng., Ph.D., and Dr. Ir. Rinaldi Munir, M.T., for dedication to teaching. Their expertise, patient guidance, and willingness to share knowledge have helped in shaping the writer's understanding of the subject.

The writer would like to express gratitude to family and friends who have been constantly supporting. Their love and support have made the completion of this research paper possible.

References

[1] "PyCubing." [Online]. Available: https://pypi.org/project/pycubing/. [Accessed: January 8, 2025]

[2] "Rubik's Cube PLL Cases." [Online]. Available: https://www.sporcle.com/games/npcds1/rubiks-cube-pll-cases. [Accessed: January 8, 2025].

[3] "Rubik's Cube." [Online]. Available: https://www.artofplay.com/products/rubiks-cube. [Accessed: January 8, 2025].

[4] "Big O Cheat Sheet: Time Complexity Chart." [Online]. Available: https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/. [Accessed: January 8, 2025].

[5] "Rubik's Cube." [Online]. Available: https://en.wikipedia.org/wiki/Rubik%27s_Cube. [Accessed: January 8, 2025].

[6] "Rubik's Cube Scrambler." [Online]. Available: https://ruwix.com/puzzle-scramble-generators/rubiks-cube-scrambler/. [Accessed: January 8, 2025].

[7] "Fridrich Method for Solving the Rubik's Cube." [Online]. Available: http://www.ws.binghamton.edu/fridrich/. [Accessed: January 8, 2025].

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 2 Januari 2025

Ahmad Wicaksono/13523121