GOD'S NUMBER OF BI-COLORED CUBES

STEVEN J. MILLER, M.TIP PHAOVIBUL, AND MUTIAN SHEN

Abstract. The Rubik's Cube, a quintessential mathematical puzzle, has long been a subject of recreational fascination and academic inquiry. God's Number for the classical 6-Colored cube has been extensively studied, and standard cases of Bi-Colored Rubik's Cube have been explored. This paper continues the exploration and extends the discussion to extreme cases in $2 \times 2 \times 2$ and $3 \times 3 \times 3$ cubes under the Quarter-Turn Metric. By employing group theory, Burnside's counting theorem, and computational algorithms such as breadth-first search (BFS) and symmetric reduction, we calculate the total number of configurations and determine God's Numbers for various Bi-Colored scenarios. However, computational limitations, particularly memory requirements, constrained our ability to analyze higher-order cubes and more complex configurations. Future developments, including coset methods and machine learning approaches, promise to overcome these challenges, enabling the exploration of larger cubes and Multi-Colored configurations with enhanced efficiency and scalability.

Contents

1. *Introduction*

The Rubik's Cube, invented in 1974 by Hungarian architect Ernő Rubik, is a three-dimensional mechanical puzzle that has become one of the most iconic and enduring challenges in recreational mathematics and popular culture. It even leads to applications in cryptography and physics([9], [17]). Comprising 26 smaller cubes, or "cubies," it forms a $3 \times 3 \times 3$ structure with six faces, each consisting of nine squares. The goal is to twist and rotate the layers of the cube to align each face with a uniform color after being scrambled. The Rubik's Cube is renowned not only for its entertainment value but also for its mathematical complexity. The puzzle's configuration space comprises over 43 quintillion ($4.3 \cdot 10^{19}$) possible arrangements, yet it is known that every configuration can be solved in 20 moves or fewer—a number famously referred to as "God's Number." This name reflects the idea of ultimate perfection and efficiency, which an all-knowing entity would achieve. The Rubik's Cube problem is formalized and extended by defining $G(n)$ and two metrics.

Definition 1.1 (God's Number of Cubes)**.** Let $G(n) = G_{1,1,1,1,1,1}(n)$ denote the God's number for an $n \times n \times n$ cube, where each of the six faces is a distinct color, defined as the maximum number of moves required to solve the cube from any scrambled state under a given metric. We denote n as the order of the cube.

Definition 1.2 (Quarter-Turn Metric)**.** The quarter-turn metric counts each 90° rotation of one face of the cube as a single move. For example, a clockwise or counterclockwise 90° turn of any face is considered one move.

Definition 1.3 (Half-Turn Metric)**.** The half-turn metric counts both 90° and 180° rotations of one face of the cube as a single move. In this metric, both quarter- and half-turns are equivalent in terms of their cost.

It is worth emphasizing that while quarter-turns and half-turns are similar operations, they differ in how they contribute to the move count under distinct metrics. As a result, God's numbers obtained in different metrics are different, and the analysis differs slightly. When $n \leq 3$, the result is well-studied due to the relatively small size of possible configurations. When $n = 2$, God's number $G(2)$ is known to be 14 by quarter turns and 11 by half turns, where a single half turn is defined as rotating any face by 180° (`https://www.jaapsch.net/puzzles/cube2.htm`). The God's number of standard Rubik's cube, which is $G(3)$, is mostly studied. By 1980, a lower bound for $G(3)$ in half turns was known to be 18, while the upper bound was around 80. For almost 30 years after that, through the unremitting exploration of mathematicians, the gap was eventually closed at 20 by the works of Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge ([13]). The timetable available on `https://www.cube20.org` is visually represented in Figure 1.

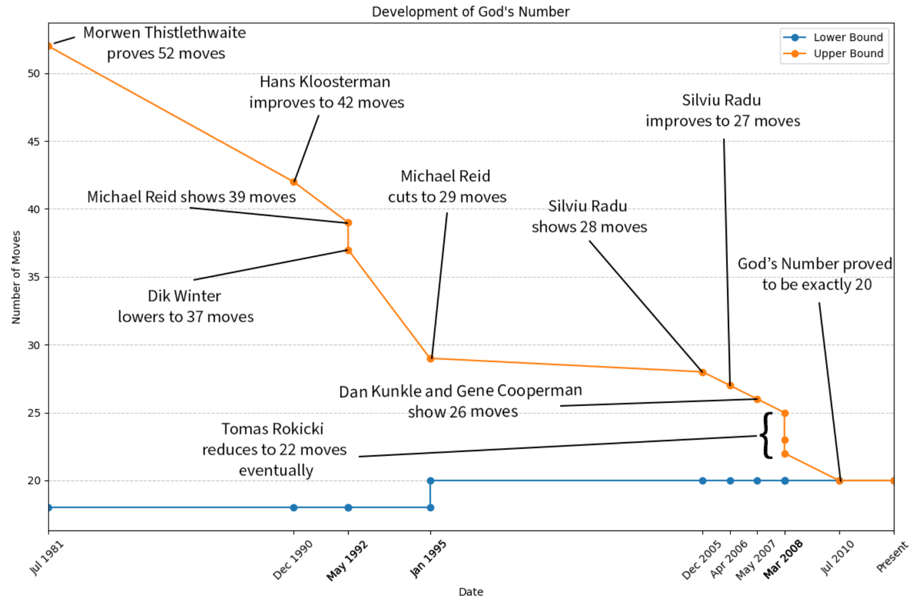

Figure 1. Development of God's Number in Half-Turn Metric (Most results were only reported electronically: see [4] for links).

The breakthrough for the quarter turns came slightly later by Tomas Rokicki and Morley Davidson ([12]). They showed that $G(3)$ in the Quarter-Turn Metric is 26 in 2014. A similar figure for the progress is shown in Figure 2.

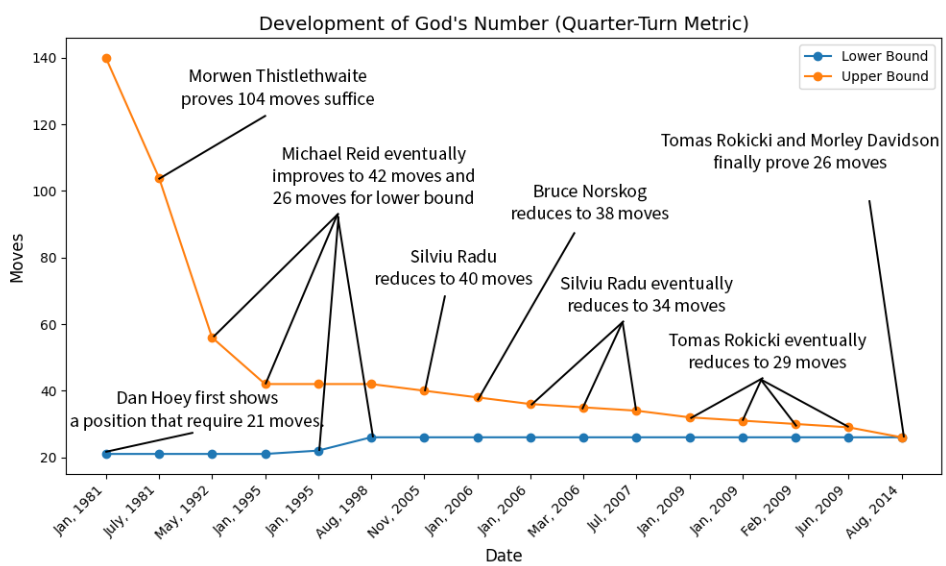

Figure 2. Development of God's Number in Quarter-Turn Metric (Most results were only reported electronically: see [11] for links).

For $G(4)$ or even larger, no result has been proven due to the large size of all possible configurations. Recently, Salkinder estimated that the God's number's growth rate is $\Omega(n^2/\log(n))$ ([15]).

This paper addresses the Bi-Colored case, which serves as a tractable but non-trivial extension of classical Rubik's Cube studies. The standard result $G_{3,3}(3)$ has been explored in Pieper's thesis ([10]). We extend the results to all possible extreme cases of the pocket cube and $G_{1,5}(3)$. We provide a systematic analysis of the Bi-Colored $2 \times 2 \times 2$ and $3 \times 3 \times 3$ cube, introducing the necessary mathematical framework and computational methods under the Quarter-Turn Metric. This work also assumes that all squares of the Rubik's Cube are solid colors. The cases are different when pictures replace solid colors.

The number of configurations reachable at each depth in the solving process forms integer sequences that reflect the underlying combinatorial structure. For example, for the $(5,1)_3$ case defined in Definition 1.7, the sequence of reachable states grows as follows: 1, 8, 76, 680, 5714, . . . , illustrating the exponential increase in states. These sequences contribute to the broader study of combinatorial integer sequences and underscore the complexity of solving asymmetric cube configurations.

Definition 1.4 (God's Number of Bi-Colored Cubes)**.** Let $G_{k,6-k}(n)$ denote God's number for an $n \times n \times n$ cube with solid colors on all squares, with k faces of one color and $6-k$ faces of a different color, defined as the maximum number of moves required to solve the cube from any scrambled state under Quarter-Turn Metric.

As Rubik's Cube can be described as a finite group where each configuration corresponds to an element of the group, and each legal move is a group operation. As the group is finite, any state can be expressed as a finite product of turns applied on the solved state. This guarantees that every configuration is a finite number of moves away from the identity element, corresponding to the solved state. Consequently, God's number is well-defined. The proof of Rubik's Cube as a group is provided in Section 2.1. Considering $G_{4,2}(n)$ and $G_{3,3}(n)$, there exist two possible colorings. To explicitly distinguish them, we make the following definitions.

Definition 1.5 (God's Number of Bi-Colored Cubes with Opposite Coloring)**.** Let $G'_{k,6-k}(n)$ denote God's number $G_{k,6-k}(n)$, where a pair of opposite faces with the same color exists.

Definition 1.6 (God's Number of Bi-Colored Cubes with Adjacent Coloring)**.** Let $G''_{k,6-k}(n)$ denote God's number $G_{k,6-k}(n)$, where k faces of one color form a connected, adjacent region, and the remaining $6-k$ faces are of a different color.

Definition 1.7 (Bi-Colored Case)**.** Let $(x,y)_n$ denote the configuration of an $n \times n \times n$ cube where x faces are colored with color C_1, and the remaining $y = 6-x$ faces are colored with color C_2.

Using results from group theory and Burnside's counting theorem, we calculated the total number of configurations for all Bi-Colored cases $(5,1)_2$, $(4,2)_2$, and $(3,3)_2$ and $(5,1)_3$. Using a breadth-first search (BFS) approach algorithm, we calculate God's number for various Bi-Colored configurations and explore how symmetry considerations affect results. The most complex result obtained is for $G_{5,1}(3)$ in Table 1. "Depth" refers to the number of quarter turns. "New States Found" refers to unique states that can be achieved at the current depth and have never been found before.

Depth	New States Found	Total States Can Be Reached
0	1	1
1	8	9
2	76	85
3	680	765
4	5714	6479
5	47558	54037
6	376614	430651
7	2646584	3077235
8	13077539	16154774
9	23709256	39864030
10	5033865	44897895
11	8505	44906400

TABLE 1. Exploration results for $(5, 1)_3$ and $G_{5,1}(3)$.

We start by recognizing the Rubik's Cube as a group following proper definitions of operations applied to it. Then, we use some related group properties for calculations of the total number of configurations from $2 \times 2 \times 2$ to $3 \times 3 \times 3$. The formulation of algorithms and results of God's numbers are introduced in Section 4.

2. *Notation and Classification*

In this section, we explore how the Rubik's Cube satisfies the properties of a group and how the number of possible configurations of both the $3 \times 3 \times 3$ and $2 \times 2 \times 2$ cubes is derived. The study of the Rubik's Cube through group theory allows us to formalize its behavior and understand the structure of these permutations. Moreover, it provides tools for solving the cube.

2.1. The Rubik's Cube as a Group

The Rubik's Cube can be modeled as a group, where each element of the group corresponds to a specific rotation of one of its faces. We give the following definitions for all possible quarter turns.

Definition 2.1 (Face Rotations)**.** The Rubik's Cube is manipulated through face rotations, each of which corresponds to a specific move. Let F, R, U, L, D, B represent 90-degree clockwise rotations of the front, right, upper, left, down, and back faces, respectively.

Definition 2.2 (Inverse Moves)**.** Each face rotation has its inverse, denoted by F', R', U', L', D', B', which represent counterclockwise rotations of the front, right, upper, left, down, and back faces, respectively.

A group operation is the sequential composition of these moves. The Rubik's Cube satisfies the four fundamental properties of a group.

- **Closure**: The composition of any two moves on the cube results in another valid cube configuration. For example, applying $F \cdot R$ produces a new permutation of the cube pieces.
- **Identity**: The solved state of the cube is the identity element of the group. Any move followed by its inverse returns the cube to this solved state. For example, $F \cdot F' = e$, where e is the identity element, representing the solved cube.

- **Invertibility**: Every move has a corresponding inverse that undoes the effect of the original move. For example, if the move R rotates the right face of the cube clockwise, then R' undoes this by rotating the same face counterclockwise.
- **Associativity**: The composition of the moves is associative; that is, for any moves A, B, C, the result of $(A \cdot B) \cdot C$ is the same as $A \cdot (B \cdot C)$. This property holds for any sequence of face rotations.

As a concrete example, consider the sequence $F \cdot U \cdot R'$, which changes the configuration of the cube. Applying their inverses $R \cdot U' \cdot F'$ returns the cube to its solved state, satisfying both the identity and the invertibility property. Some key properties of a group, consequently, can be used to analyze Rubik's cube, develop solving algorithms, and understand its complexities.

- **Commutativity**: Some sequences of moves on the cube commute, meaning that the order of applying them does not change the outcome. For example, rotating the front face followed by rotating the back face often results in the same configuration regardless of the order.
- **Cyclic Groups**: Each face of the cube generates a cyclic group of order 4. That is, rotating any face by 90 degrees four times returns the cube to its original configuration. The same principle applies to face inverses, where applying $F^4 = e$, which means that four 90-degree rotations on the front face restore the cube to its initial state.
- **Conjugacy**: In solving strategies, conjugates play an important role. A sequence of moves A, followed by a different move B, and then the inverse of A, is known as a conjugate. Conjugates allow solvers to manipulate specific parts of the cube while leaving other areas unchanged. For example, the sequence $F \cdot R \cdot F'$ applies a targeted transformation to the cube while preserving the rest of its structure.

These properties are exploited in various solving methods, allowing for more efficient algorithms that minimize the number of moves required to solve the cube. Furthermore, these properties help explain the cube symmetry and provide insight into how the group structure governs its behavior.

2.2. General Methods for Solving based on Group Property

Based on the group properties of the Rubik's Cube, a widely-used method for solving it, called the layer-by-layer method, involves solving each layer sequentially while preserving the already solved ones. The layer-by-layer method systematically utilizes the group structure of the cube by progressively reducing the configuration space through a sequence of stabilizer subgroups. Each step corresponds to solving a specific layer, mathematically represented by restricting the group G to smaller subgroups $H_1 \supset H_2 \supset \cdots \supset H_k$. Key group properties, such as closure, parity constraints, and commutators, ensure that solved layers remain invariant during the manipulation of unsolved parts. Conjugation and coset operations further localize transformations to targeted pieces while preserving the cube's overall parity. By Lagrange's theorem, the size of each stabilizer subgroup is a divisor of G, ensuring convergence to the solved state. Details of the specific steps can be found in David Singmaster's work. Some other general methods include the CFOP method by Jessica Fridrich and others in the 1980s ([16]) and the Petrus Method ([7]). Despite these general methods not necessarily being efficient, they are foundational in understanding the mathematical structure of the Rubik's Cube and provide a systematic framework for solving it.

2.3. Configurations and Counting of Order 3 Cube

The number of possible configurations of the $n \times n \times n$ Rubik's Cube is determined by the permutations and orientations of its corner and edge pieces. For any cube, there are always 8 corner pieces with three possible colors and $12(n-2)$ edge pieces with 2 possible colors. When limiting the discussion for $n \leq 3$ in the paper, only the orientations and positions of the corner and edge pieces determine each distinct configuration of Rubik's cube. Hence, some definitions and counting techniques are introduced as follows.

2.3.1. Cubies, Permutations, and Orientations

Definition 2.3 (Cubie)**.** A **cubie** is a block that occupies one position on the Rubik's Cube and contains solid colored stickers.

Definition 2.4 (Corner Cubie)**.** A **corner cubie** is a type of cubie with three stickers. It occupies one of the 8 corner positions on the Rubik's Cube and is denoted as x_i where $i \in [1, 8]$.

Definition 2.5 (Edge Cubie)**.** An **edge cubie** is a type of cubie with two stickers. It occupies one of the $12(n-2)$ edge positions on an $n \times n \times n$ Rubik's Cube and is denoted as y_i where $i \in [1, 12(n-2)]$.

Definition 2.6 (Orientation of a Cubie)**.** The **orientation** of a cubie refers to its rotational state within its position on the Rubik's Cube. For a corner cubie, there are three possible orientations, represented by the set $x_i = \mathbb{Z}/3\mathbb{Z}$. For an edge cubie, there are two possible orientations, represented by the set $y_i = \mathbb{Z}/2\mathbb{Z}$.

Definition 2.7 (Permutation of a Cubie)**.** The **permutation** of a cubie refers to its position relative to other cubies on the Rubik's Cube. In a standard Rubik's Cube, the permutation is represented by an ordered list of $x_1, \ldots, x_8$ and $y_1, \ldots, y_{12}$.

The orientation specifies how the stickers on a cubie are aligned relative to the solved state. For example, a corner cubie with three stickers can rotate within its position in three distinct ways. Similarly, an edge cubie with two stickers has only two possible orientations: aligned or flipped. The permutation of cubies determines the arrangement of all cubies on the cube. For corner cubies, the current configuration can be represented by assigning an element in $\mathbb{Z}/3\mathbb{Z}$ to each cubie. For example, a configuration of the eight corner cubies can be expressed as $x_1, x_2, \ldots, x_8 \in \mathbb{Z}/3\mathbb{Z}$. Together, orientation and permutation fully specify the current state of the cube.

2.3.2. Corner Permutations and Orientations

For the 8 corner cubies, the three colors of each can be labeled $\mathbb{Z}/3\mathbb{Z}$. An example of labeling on one face of the $3 \times 3 \times 3$ Rubik's cube is shown in Figure 3 for illustration.

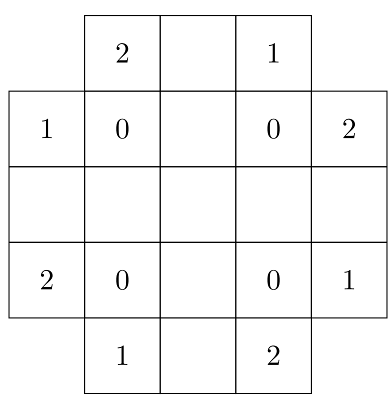

FIGURE 3. Front face of $3 \times 3 \times 3$ Rubik's Cube with labeled corner orientations.

When one move is applied to the cube, F as an example, the labeling would be the same as that in Figure 3, but the change of relative positions leads to a different configuration for the cube. This change is demonstrated in Figure 4.

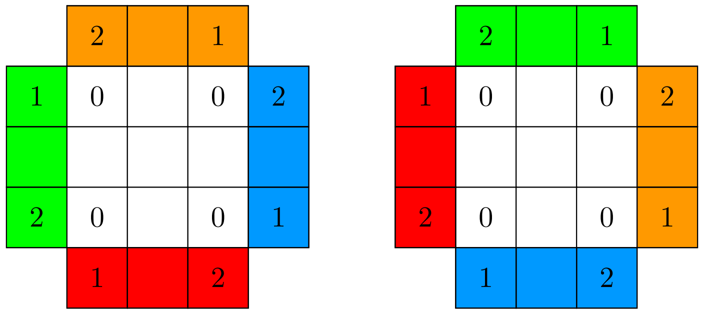

FIGURE 4. Left: original state, Right: after F move.

Meanwhile, changing the label, representing the change in orientation, would lead to another configuration. Taking the top right corner as an example, both the relative positions of the corners and the orientation change as shown in Figure 5.

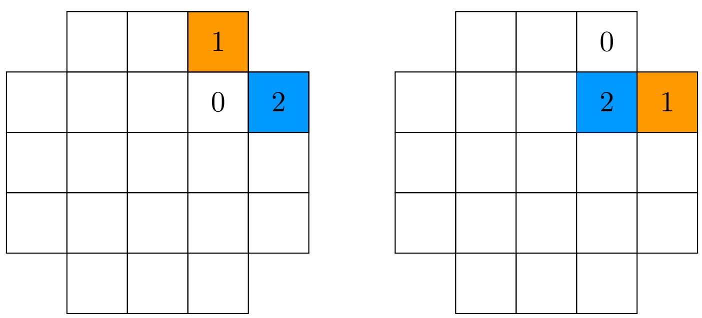

FIGURE 5. Left: original state, Right: after R move.

The total number of unique configurations of the corner cubies, as a result, is $8! \cdot 3^8$ considering the relative positions of $x_1, \ldots, x_8$ and the orientation of each of them. However, the cubie would be unsolvable when one corner cubie is rotated. One important result states that $\sum_{i=1}^{8} x_i \equiv 0 \pmod 3$. The proof is straightforward. Consider the cube of the solved status. The result holds naturally. A right move, as shown in Figure 5, changes the value of the four corner cubies by $1, 2, 1, 2$, again satisfying the result. Using similar arguments, each move maintains the change a multiple of 3, and the result holds. Hence, the total number of corner configurations is calculated in Equation 1.

$$8! \cdot \frac{3^8}{3} \;=\; 8! \cdot 3^7. \tag{1}$$

2.3.3. Edge Permutations and Orientations

For the 12 edge cubies of the $3 \times 3 \times 3$ Rubik's Cube, each cubie is formally defined in Definition 2.5 2.6. Similarly to corner cubies, both the orientation and permutations of edge cubies determine the total number of configurations of the Rubik's cube. Consider the R move as an example shown in Figure 6.

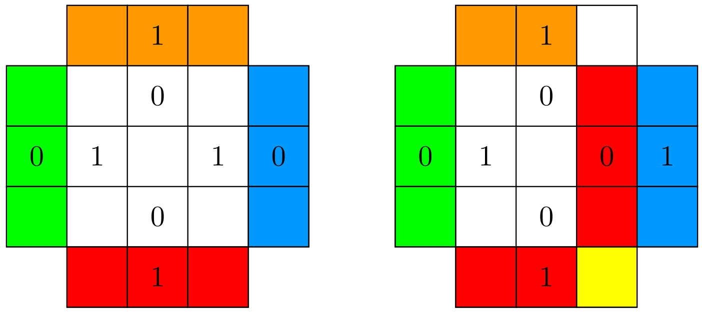

FIGURE 6. Left: original state, Right: after R move.

When R move is placed, both the relative positions of y_i and their labeling change, leading to a new configurations. Naitively like the discussion in corner part, the total number of unique configurations of the edge cubies is initially $12! \cdot 2^{12}$. A similar result also holds such that $\sum_{i=1}^{12} y_i \equiv 0 \pmod 2$. Starting from the solved state, the sum is zero. Each basic move (like F, R, etc.) flips an even number of edge cubies. For example, the F move flips the orientations of the four front face edges, but since flipping an edge orientation is equivalent to adding 1 modulo 2, and $1+1+1+1=0$ mod 2, the total remains even. Therefore, after any sequence of valid moves, the sum remains congruent to zero modulo 2. The total number of edge configurations is thus obtained in Equation 2.

$$12! \cdot \frac{2^{12}}{2} \;=\; 12! \cdot 2^{11}. \tag{2}$$

2.3.4. Parity condition

In combinatorics, a permutation is an arrangement of elements in a set. The parity of a permutation can either be even or odd.

(1) A permutation is *even* if it can be achieved using an even number of swaps between elements.
(2) A permutation is *odd* if it requires an odd number of swaps.

For instance, the permutation $(1\,2\,3) \to (2\,1\,3)$ is odd because it involves swapping 1 and 2, while $(1\,2\,3) \to (3\,1\,2)$ is even because it can be achieved with two swaps. In the context of the Rubik's Cube, every face rotation is composed of cycles that involve moving pieces in a way that corresponds to an even permutation. Specifically, a face rotation affects four corner pieces, rotating them in a cycle. This cycle is an even permutation since it can be represented as a 4-cycle, which is an even permutation. Similarly, a face rotation affects four edge pieces, also rotating them in a cycle, which is an even permutation. Since both the corner permutation and the edge permutation resulting from a face rotation are even, any sequence of face rotations (legal moves) always results in permutations where the parities of the corners and edges match. It is impossible to achieve, through legal moves, a state where the corners are in an even permutation while the edges are in an odd permutation, or vice versa. Thus, only half of the configurations in earlier parts are valid, as the parity of the corners and edges must match. Thus, the total number of valid configurations is calculated in Equation 3.

$$\frac{8! \cdot 3^7 \cdot 12! \cdot 2^{11}}{2} \;=\; 43,252,003,274,489,856,000. \tag{3}$$

2.4. Configurations and Counting of Order 2 Cube

The $2 \times 2 \times 2$ cube is a simplified version of the $3 \times 3 \times 3$ cube, consisting only of 8 corner pieces. The number of possible configurations is determined by the permutations and orientations of the corner cubies, without edge pieces.

2.4.1. Corner Permutations and Orientations

The 8 corner cubies can be permuted among the 8 corner positions in $8! = 40,320$ ways. Each corner cubie can be oriented in 3 ways, but the orientation of the eighth corner depends on the first seven, resulting in $3^7 = 2,187$ valid orientations. However, without the center cubie to fix the orientation of each face of the cube, which happens to all even cubes, the rotation of the entire cube such as $U \cdot D$ in the $2 \times 2 \times 2$ cube preserves the original configuration. Since each face can be placed as the front with 4 possibilities. The total number of configurations is reduced by a factor of 24, and the result for the pocket cube is shown in Equation 4.

$$8! \cdot 3^7 \cdot \frac{1}{6 \cdot 4} \;=\; 3,674,160. \tag{4}$$

3. *Bi-Colored Cases of Order 2 Cube*

While the god's number of the 6-Colored $2 \times 2 \times 2$ cube has been fully examined, some cases with fewer colors remain unclear. The total number of configurations, with and without the consideration of the symmetry property brought by the lack of a center cubie at each face, is discussed. For better visualization, denote $C_1 = White$, $C_2 = Yellow$.

3.1. $(5,1)_2$ Case

The case represents 5 faces in color C_1, while the last face is in a different color, C_2. The labeling and color arrangement of the sample in the solved state is shown in Figure 7.

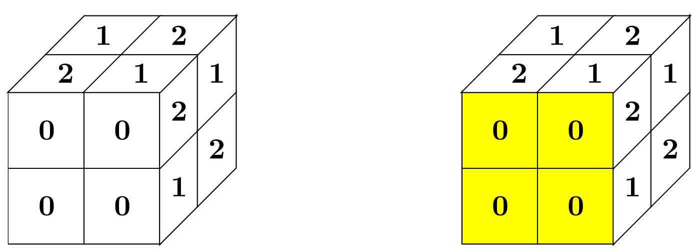

FIGURE 7. $(5,1)_2$ cube faces F, R, U and cube faces B, L, D.

The figure on the left demonstrates the front, right, and top faces of the cube, where all faces are colored white. The figure on the right represents the back, left, and down faces of the cube, where only the back face is colored yellow. The total number of configurations, without considering the symmetry property mentioned in the general case of the $2 \times 2 \times 2$ cube, is calculated in Equation 5.

$$\frac{8!}{4! \cdot 4!} \cdot 3^4 \;=\; 5670. \tag{5}$$

Note that the orientation constraints, which cause the division by 3 in the previous part, disappear when different types of corner cubies violate the pure $\mathbb{Z}/3\mathbb{Z}$ labeling.

3.2. $(4,2)_2$ Case

The case also represents a 2−color case, where 4 faces in color C_1 while the rest 2 faces are in a different color C_2. The first scenario is when two faces of C_2 are opposite, which is visually presented in Figure 8. The result corresponds to $G_{4,2}(2)'$.

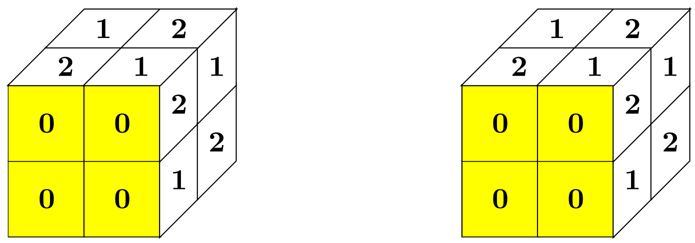

FIGURE 8. Scenario 1: $(4,2)_2$ cube faces F, R, U and cube faces B, L, D.

The structure is the same as the $(5,1)_2$ case, but the front and back faces are now both colored yellow. Without the symmetry discussion, when all 8 corner cubies are colored exactly the same, the total number of configurations is shown in Equation 6.

$$\frac{8!}{8!} \cdot \frac{3^8}{3} \;=\; 2187. \tag{6}$$

The second scenario occurs when the faces of C_2 are adjacent. A sample figure showing its solved states is provided in Figure 9. The result corresponds to $G_{4,2}(2)''$.

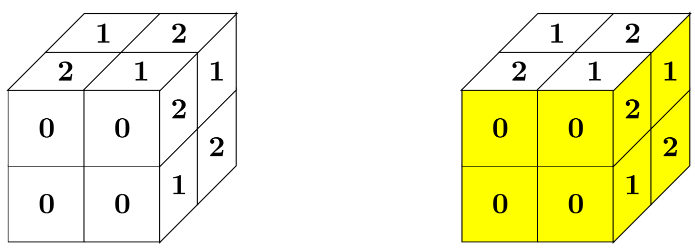

FIGURE 9. Scenario 2: $(4,2)_2$ cube faces F, R, U and cube faces B, L, D.

Now, the left and back faces are both colored yellow. There are three types of corner cubies.

- 2 corner cubies with (C_1, C_1, C_1).
- 4 corner cubies with (C_1, C_1, C_2).
- 2 corner cubies with (C_1, C_2, C_2).

It leads to the calculation of the total number of configurations in Equations 7.

$$\frac{8!}{2!\cdot 2!\cdot 4!} \cdot 3^6 \;=\; 306180. \tag{7}$$

3.3. $(3,3)_2$ Case

The case is also for Bi-Colored case, where 3 faces in color C_1 and 3 faces in color C_2. There are also 2 different scenarios. The first scenario is when one pair of faces in C_1 is opposite(Figure 10). The corresponding God's number is $G'_{3,3}(2)$.

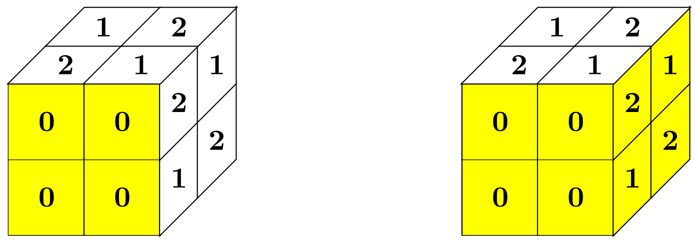

FIGURE 10. Scenario 1: $(3,3)_2$ cube faces F, R, U and cube faces B, L, D.

The front, left, and back faces are colored in C_2. The left and right faces are in the opposite positions but colored differently. There are two types of corner cubies.

- 4 corner cubies with (C_1, C_1, C_2).
- 4 corner cubies with (C_1, C_2, C_2).

The number of configurations, as a result, is obatined in Equation 8.

$$\frac{8!}{4! \cdot 4!} \cdot \frac{3^8}{3} \;=\; 153090. \tag{8}$$

The second scenario is when all faces that are colored the same are adjacent to each other. Another sample figure is drawn in Figure 11. The corresponding God's number is $G''_{3,3}(2)$.

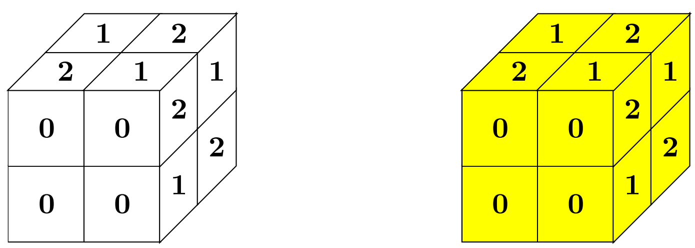

FIGURE 11. Scenario 2: $(3,3)_2$ cube faces F, R, U and cube faces B, L, D.

The back, left, and down faces are colored in C_2 while the rest faces are colored in C_1. It changes the corner cubies into the following four types.

- 1 corner cubies with (C_1, C_1, C_1).
- 3 corner cubies with (C_1, C_1, C_2).
- 3 corner cubies with (C_1, C_2, C_2).
- 1 corner cubies with (C_2, C_2, C_2).

The total number of configurations is derived in Equation 9.

$$\frac{8!}{3! \cdot 3! \cdot 1! \cdot 1!} \cdot 3^6 \;=\; 816480. \tag{9}$$

3.4. Permutations and Symmetry

While exploring the less colored pocket cube, it is worth noting that the formula for calculating the total number of configurations varies depending on the labeling of each cubie. Similarly, when considering the symmetry brought by the lack of center cubies on each face to fix position, the flexibility for the entire cube to rotate as a whole makes the counting complicated. Simple strategies such as division by 24 no longer work. Burnside's counting theorem, also known as the Cauchy–Frobenius lemma, should be applied for more careful analysis. Let G be a finite group acting on a set X. The number of distinct orbits of the action of G on X is given in Equation 10.

$$|X/G| \;=\; \frac{1}{|G|}\sum_{g\in G}|X^g|. \tag{10}$$

where $|G|$ is the order of the group G,$X^g = \{x \in X : g \cdot x = x\}$ is the set of elements in X that are fixed by the group element g, $|X^g|$ denotes the cardinality of the set X^g.The number of distinct orbits is the average number of elements fixed by the group elements. Using such an approach, it becomes possible to eliminate symmetric cases. The results for all three 2−color cases are listed in Table 2.

Case	**Scenario**	**Number of Configurations**
$(5,1)_2$	–	258
$(4,2)_2$	Scenario 1	102
	Scenario 2	12,879
$(3,3)_2$	Scenario 1	6,441
	Scenario 2	34,032

TABLE 2. Configurations for Bi-Colored Cubes.

4. Bi-Colored Cases of Order 3 Cube

4.1. $(5,1)_2$ Case

Followed by the definitions in 2.4, 2.5, and 1.7, we define a $3 \times 3 \times 3$ cube with 5 faces in one color C_1 while the remaining face in another color C_2. Using a similar treatment as in Section 3 3, we can let $C_1 = White$, $C_2 = Yellow$ for better visualization. The only scenario is represented in solved states in Figure 12.

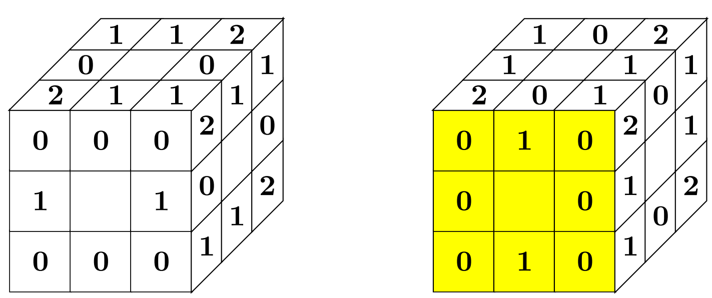

FIGURE 12. $(5,1)_3$ cube faces F, R, U and cube faces B, L, D.

The total number of configurations, with the edge cubies added, is displayed in Equation 11.

$$\frac{8!}{4!\cdot 4!}\cdot 3^4 \cdot \frac{12!}{4!\cdot 8!}\cdot 2^4 \;=\; 44906400. \tag{11}$$

5. *God's Number Results*

5.1. $G_{5,1}(2), G_{4,2}(2)$, and $G_{3,3}(2)$

All of our explorations are built on the Quarter-Turn Metric. The exploration of God's number is examined through a brute force approach due to the relatively limited size of possibilities. The breadth first search approach (BFS) is used to explore possible new configurations until no new states can be found. The search result when symmetry is not considered is stated in Table 3, Table 4, and Table 5. In the table, "Depth" refers to the number of quarter turns applied on the solved state. "New States Found" refers to unique states that can be achieved at the current depth and have never been found before. "Total States Can Be Reached" refers to the total number of unique states that can be reached within the current depth.

Depth	New States Found	Total States Can Be Reached
0	1	1
1	8	9
2	60	69
3	332	401
4	1343	1744
5	2988	4732
6	932	5664
7	6	5670

TABLE 3. Results of $G_{5,1}(2)$ without symmetric reduction.

Scenario	Depth	New States Found	Total States Can Be Reached
Scenario 1	0	1	1
	1	4	5
	2	26	31
	3	110	141
	4	372	513
	5	684	1197
	6	816	2013
	7	150	2163
	8	24	2187
Scenario 2	0	1	1
	1	12	13
	2	106	119
	3	776	895
	4	4461	5356
	5	19832	25188
	6	64030	89218
	7	124374	213592
	8	87032	300624
	9	5556	306180

TABLE 4. Combined Table for Results of $G_{4,2}(2)$ without symmetric reduction.

Scenario	Depth	New States Found	Total States Can Be Reached
Scenario 1	0	1	1
	1	10	11
	2	93	104
	3	694	798
	4	4055	4853
	5	17140	21993
	6	50797	72790
	7	63472	136262
	8	16636	152898
	9	192	153090
Scenario 2	0	1	1
	1	12	13
	2	99	112
	3	648	760
	4	3663	4423
	5	17580	22003
	6	67851	89854
	7	199812	289666
	8	340086	629752
	9	178168	807920
	10	8560	816480

TABLE 5. Combined Table for Results of $G_{3,3}(2)$ without symmetric reduction.

The maximum depth is God's number as it represents the most complicated configuration. When symmetry is considered and matches the real pocket cube, it makes the total number of configurations significantly fewer. The symmetric reduction is achieved through a conversion from configurations to numbers. Since each configuration in a pocket cube can repeat at most 24 times, the lowest conversion result is taken for hashing and subsequent comparisons. With the symmetric reduction, all results are shown below in Table 6, Table 7, and Table 8.

Depth	New States Found	Total States Can Be Reached
0	1	1
1	2	3
2	5	8
3	21	29
4	66	95
5	121	216
6	41	257
7	1	258

TABLE 6. Results of $G_{5,1}(2)$.

Scenario	Depth	New States Found	Total States Can Be Reached
Scenario 1 for $G'_{4,2}(2)$	0	1	1
	1	1	2
	2	2	4
	3	5	9
	4	17	26
	5	31	57
	6	37	94
	7	7	101
	8	1	102
Scenario 2 for $G''_{4,2}(2)$	0	1	1
	1	4	5
	2	16	21
	3	58	79
	4	227	306
	5	855	1161
	6	2634	3795
	7	5192	8987
	8	3656	12643
	9	236	12879

Table 7. Combined Table for Results of $G_{4,2}(2)$.

Scenario	Depth	New States Found	Total States Can Be Reached
Scenario 1 for $G'_{3,3}(2)$	0	1	1
	1	5	6
	2	14	20
	3	52	72
	4	210	282
	5	741	1023
	6	2086	3109
	7	2630	5739
	8	694	6433
	9	8	6441
Scenario 2 for $G''_{3,3}(2)$	0	1	1
	1	2	3
	2	9	12
	3	40	52
	4	178	230
	5	746	976
	6	2801	3777
	7	8300	12077
	8	14168	26245
	9	7429	33674
	10	358	34032

Table 8. Combined Table for Results of $G_{3,3}(2)$.

5.2. $G_{5,1}(3)$

By our calculation in Section 4, we can still manage to explore $G_{5,1}(3)$ in Bi-Colored $3 \times 3 \times 3$ cube given the total number of configurations are not too large. The center cubie on each face fixes the relative position and removes the need for symmetric reduction. Hence, the algorithm, with the addition of edge cubies, manages to produce results for $G_{5,1}(3)$ as shown in Table 1. When the case becomes slightly more complex, such as $(4,2)_3$ and $(3,3)_3$, the drastic increase in the total number of configurations forbids direct numerical results. The simpler case of $(4,2)_3$ requires at least $40G$ memory to run.

6. Conclusion and Future Development

This study identifies the complexities of solving Bi-Colored Rubik's Cube configurations, enhancing our understanding of their unique properties. These findings contribute to the larger effort to categorize God's numbers for nonstandard cases, paving the way for further exploration of multicolored and higher order cubes. The results are summarized in Table 9.

God's Number Symbol	Value
$G_{5,1}(2)$	7
$G'_{4,2}(2)$	8
$G''_{4,2}(2)$	9
$G'_{3,3}(2)$	9
$G''_{3,3}(2)$	10
$G_{5,1}(3)$	11

TABLE 9. Combined Results of God's Number in Bi-Colored Cubes in Quarter-Turn Metric.

The computational effort required for this project increased significantly as the size of the cube grew and as the complexity of the Bi-Colored cases intensified. It is expected to have at least $40G$ memory for the simpler case in $(4,2)_3$. It might be possible to apply the algorithms directly for $G_{4,2}(3)$ and $G_{3,3}(3)$ with more computing resources. Such an approach should be modified when cases become more complex, such as a higher order of Rubik's Cube or more than 2 colors. In Pieper's thesis, one case of $(3,3)_3$ was explored and the total configurations found are shown to be 10,344,206,272, but the other case remains unknown due to the limit of computational power([10]).

One direction of future improvements is the use of coset methods. By focusing on coset representatives and stabilizer subgroups, such a method significantly reduces the need to store each different configuration and allows for much faster computation([9]). This method was also used in the discovery of $G(3)$ ([12], [13]). Another direction of future works is the analysis of different shapes of puzzles in the style of Rubik's Cube. Some potential extensions include Pyraminx, Megaminx, Skewb, and Fenghuolun. Some interesting results have been shown such as the God's number of Pyraminx Duo ([5]).

Limited by the computing power, the deterministic method to compute God's number is fairly difficult and even impossible when the cube size n becomes large. The machine learning method, on the other hand, provides an alternative to efficiently solving Rubik's cube. Some results have proven the effectiveness of such an approach. Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi built a model based on deep learning and reinforcement, which optimally solves 60.3% of $3 \times 3 \times 3$ cubes ([1]). More subsequent works on machine learning, such as entropy modeling ([2]), Autodidactic Iteration as one reinforcement learning approach ([8]), and various deep learning approaches ([6], [14], [3]) have demonstrated significant potential in solving Rubik's cubes, particularly for standard $3 \times 3 \times 3$ cases. Their application in Bi-Colored cases and larger size n is an exciting unexplored territory.

References

[1] Forest Agostinelli et al. "Solving the Rubik's cube with deep reinforcement learning and search". In: *Nature Machine Intelligence* 1.8 (2019), pp. 356–363.

[2] BV Amrutha and Ramamoorthy Srinath. "Deep Learning Models for Rubik's Cube with Entropy Modelling". In: *ICDSMLA 2020: Proceedings of the 2nd International Conference on Data Science, Machine Learning and Applications*. Springer. 2022, pp. 35–43.

[3] Sebastiano Corli et al. "Solving rubik's cube via quantum mechanics and deep reinforcement learning". In: *Journal of Physics A: Mathematical and Theoretical* 54.42 (2021), p. 425302.

[4] *Cube20.org*. Accessed: 2024-12-02. URL: `https://www.cube20.org/`.

[5] *Jaap's Puzzle Page: Pyraminx Duo*. Accessed: 2024-12-02. URL: `https://www.jaapsch.net/puzzles/pyraduo.htm`.

[6] Colin G Johnson. "Solving the Rubik's cube with stepwise deep learning". In: *Expert Systems* 38.3 (2021), e12665.

[7] *Lars Petrus' Method*. Accessed: 2024-12-02. URL: `https://lar5.com/cube/`.

[8] Stephen McAleer et al. "Solving the Rubik's cube without human knowledge". In: *arXiv preprint arXiv:1805.07470* (2018).

[9] Christophe Petit and Jean-Jacques Quisquater. *Rubik's for cryptographers*. Cryptology ePrint Archive, Paper 2011/638. 2011. URL: `https://eprint.iacr.org/2011/638`.

[10] Taylor Pieper. "Complexities of Bi-Colored Rubik's Cubes". Undergraduate Thesis. Undergraduate Honors Thesis Collection, 2017.

[11] *Quarter Turn Metric (QTM)*. Accessed: 2024-12-02. URL: `https://www.cube20.org/qtm/`.

[12] Tomas Rokicki. "Towards God's Number for Rubik's Cube in the Quarter-Turn Metric". In: *The College Mathematics Journal* 45.4 (2014), p. 242. ISSN: 07468342, 19311346. URL: `https://www.jstor.org/stable/10.4169/college.math.j.45.4.242` (visited on 11/30/2024).

[13] Tomas Rokicki et al. "The Diameter of the Rubik's Cube Group Is Twenty". In: *SIAM Review* 56.4 (2014), pp. 645–670. DOI: `10.1137/140973499`. eprint: `https://doi.org/10.1137/140973499`. URL: `https://doi.org/10.1137/140973499`.

[14] M Mahindra Roshan et al. "Towards efficiently solving the rubik's cube with deep reinforcement learning and recursion". In: *E3S Web of Conferences*. Vol. 491. EDP Sciences. 2024, p. 01009.

[15] Daniel Salkinder. "$n \times n \times n$ Rubik's Cubes and God's Number". In: *arXiv preprint arXiv:2112.08602* (2021).

[16] David Singmaster. *Notes on Rubik's Magic Cube*. Enslow Publishers, 1981.

[17] Da-Xing Zeng et al. "Overview of Rubik's cube and reflections on its application in mechanism". In: *Chinese Journal of Mechanical Engineering* 31 (2018), pp. 1–12.

A. Code Appendix

The exploration of God's number and possible states that can be reached at each step is done by the following C++ algorithm with BFS approach. Despite only Bi-Colored cases are discussed earlier, the code is written in 6-bases that can accommodate any initial state with fewer or equal to 6 colors in total. The algorithm, once changing the initial cube labeling, will produce all results like the table earlier until all possible configurations have been explored. The `min_symmetry` function is used to transform each configuration to the lowest possible number representation in 24 possible directions to view the pocket cube, which eliminates all the symmetric configurations as a result. When the function `min_symmetry` is removed, the result will recover the cases where no symmetry is involved like the first few God's number results. All codes used are provided below:

A.1. Codes of Order 2 Cube

```
#include <algorithm>
#include <array>
#include <fstream>
#include <iostream>
#include <queue>
#include <unordered_set>
#include <vector>

// define 6 colors in total
enum colors { WHITE = 0, YELLOW = 1, RED = 2,
    ORANGE = 3, GREEN = 4, BLUE = 5 };

// define the structure of corners and 2 by 2 by 2 cube

// The corners are labled as 0 at UFR, 1 at UBR, 2 at UBL,
//3 at UFL, 4 at DFR, 5 at DBR, 6 at DBL, 7 at DFL
using Corner = std::array<colors, 3>;

using Cube = std::array<Corner, 8>;

// define Front Rotation
Cube rotate_Front(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube[0];
    Corner temp3 = cube[3];
    Corner temp4 = cube[4];
    Corner temp7 = cube[7];
    new_state[0] = { temp3[2], temp3[0], temp3[1] };
    new_state[3] = { temp7[1], temp7[2], temp7[0] };
    new_state[4] = { temp0[1], temp0[2], temp0[0] };
    new_state[7] = { temp4[2], temp4[0], temp4[1] };
    return new_state;
}
Cube rotate_Front_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube[0];
    Corner temp3 = cube[3];
    Corner temp4 = cube[4];
    Corner temp7 = cube[7];
    new_state[0] = { temp4[2], temp4[0], temp4[1] };
    new_state[3] = { temp0[1], temp0[2], temp0[0] };
    new_state[4] = { temp7[1], temp7[2], temp7[0] };
    new_state[7] = { temp3[2], temp3[0], temp3[1] };
```

```
    return new_state;
}

// define Right Rotation

Cube rotate_Right(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube[0];
    Corner temp1 = cube[1];
    Corner temp4 = cube[4];
    Corner temp5 = cube[5];
    new_state[0] = { temp4[1], temp4[2], temp4[0] };
    new_state[1] = { temp0[2], temp0[0], temp0[1] };
    new_state[4] = { temp5[2], temp5[0], temp5[1] };
    new_state[5] = { temp1[1], temp1[2], temp1[0] };
    return new_state;
}

Cube rotate_Right_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube[0];
    Corner temp1 = cube[1];
    Corner temp4 = cube[4];
    Corner temp5 = cube[5];
    new_state[0] = { temp1[1], temp1[2], temp1[0] };
    new_state[1] = { temp5[2], temp5[0], temp5[1] };
    new_state[4] = { temp0[2], temp0[0], temp0[1] };
    new_state[5] = { temp4[1], temp4[2], temp4[0] };
    return new_state;
}

Cube rotate_Back(Cube cube) {
    Cube new_state = cube;
    Corner temp1 = cube[1];
    Corner temp2 = cube[2];
    Corner temp5 = cube[5];
    Corner temp6 = cube[6];
    new_state[1] = { temp5[1], temp5[2], temp5[0] };
    new_state[2] = { temp1[2], temp1[0], temp1[1] };
    new_state[5] = { temp6[2], temp6[0], temp6[1] };
    new_state[6] = { temp2[1], temp2[2], temp2[0] };
    return new_state;
}

Cube rotate_Back_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp1 = cube[1];
    Corner temp2 = cube[2];
    Corner temp5 = cube[5];
    Corner temp6 = cube[6];
```

```
    new_state[1] = { temp2[1], temp2[2], temp2[0] };
    new_state[2] = { temp6[2], temp6[0], temp6[1] };
    new_state[5] = { temp1[2], temp1[0], temp1[1] };
    new_state[6] = { temp5[1], temp5[2], temp5[0] };
    return new_state;
}

Cube rotate_Left(Cube cube) {
    Cube new_state = cube;
    Corner temp2 = cube[2];
    Corner temp3 = cube[3];
    Corner temp6 = cube[6];
    Corner temp7 = cube[7];
    new_state[2] = { temp6[1], temp6[2], temp6[0] };
    new_state[3] = { temp2[2], temp2[0], temp2[1] };
    new_state[6] = { temp7[2], temp7[0], temp7[1] };
    new_state[7] = { temp3[1], temp3[2], temp3[0] };
    return new_state;
}

Cube rotate_Left_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp2 = cube[2];
    Corner temp3 = cube[3];
    Corner temp6 = cube[6];
    Corner temp7 = cube[7];
    new_state[2] = { temp3[1], temp3[2], temp3[0] };
    new_state[3] = { temp7[2], temp7[0], temp7[1] };
    new_state[6] = { temp2[2], temp2[0], temp2[1] };
    new_state[7] = { temp6[1], temp6[2], temp6[0] };
    return new_state;
}

Cube rotate_Top(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube[0];
    Corner temp1 = cube[1];
    Corner temp2 = cube[2];
    Corner temp3 = cube[3];
    new_state[0] = { temp3[0], temp3[1], temp3[2] };
    new_state[1] = { temp0[0], temp0[1], temp0[2] };
    new_state[2] = { temp1[0], temp1[1], temp1[2] };
    new_state[3] = { temp2[0], temp2[1], temp2[2] };
    return new_state;
}

Cube rotate_Top_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube[0];
    Corner temp1 = cube[1];
```

```
    Corner temp2 = cube[2];
    Corner temp3 = cube[3];
    new_state[0] = { temp1[0], temp1[1], temp1[2] };
    new_state[1] = { temp2[0], temp2[1], temp2[2] };
    new_state[2] = { temp3[0], temp3[1], temp3[2] };
    new_state[3] = { temp0[0], temp0[1], temp0[2] };
    return new_state;
}

Cube rotate_Down(Cube cube) {
    Cube new_state = cube;
    Corner temp4 = cube[4];
    Corner temp5 = cube[5];
    Corner temp6 = cube[6];
    Corner temp7 = cube[7];
    new_state[4] = { temp7[0], temp7[1], temp7[2] };
    new_state[5] = { temp4[0], temp4[1], temp4[2] };
    new_state[6] = { temp5[0], temp5[1], temp5[2] };
    new_state[7] = { temp6[0], temp6[1], temp6[2] };
    return new_state;
}

Cube rotate_Down_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp4 = cube[4];
    Corner temp5 = cube[5];
    Corner temp6 = cube[6];
    Corner temp7 = cube[7];
    new_state[4] = { temp5[0], temp5[1], temp5[2] };
    new_state[5] = { temp6[0], temp6[1], temp6[2] };
    new_state[6] = { temp7[0], temp7[1], temp7[2] };
    new_state[7] = { temp4[0], temp4[1], temp4[2] };
    return new_state;
}

// Convert the cube to a numerical representation for comparison
long long cubeToNumber(const Cube& cube) {
    long long number = 0;
    for (const auto& corner : cube) {
        for (const auto& color : corner) {
            number = number * 6 + color;   // Base-6 number representation
        }
    }
    return number;
}

Cube rotate_without_change(Cube cube) {
    Cube temp = cube;
    temp[0] = cube[1];
    temp[1] = cube[2];
```

```
    temp[2] = cube[3];
    temp[3] = cube[0];
    temp[4] = cube[5];
    temp[5] = cube[6];
    temp[6] = cube[7];
    temp[7] = cube[4];
    return temp;
}

// Function to rotate the cube so that a specific face becomes the top
Cube changeTopColor(Cube cube, colors newTop) {
    Cube rotated = cube;

    switch (newTop) {
    case WHITE:
        // No rotation needed
        break;
    case YELLOW:
        // Rotate 180 degrees around the x-axis (top to bottom)
        return rotate_Left(rotate_Right_Inverse(
            rotate_Left(rotate_Right_Inverse(cube))));
    case RED:
        return rotate_Left_Inverse(rotate_Right(cube));
    case ORANGE:
        return rotate_Right_Inverse(rotate_Left(cube));
    case GREEN:
        return rotate_Back_Inverse(rotate_Front(cube));
    case BLUE:
        return rotate_Front_Inverse(rotate_Back(cube));
    }

    return rotated;
}


long long min_symmetry(Cube cube) {
    long long minNumber = std::numeric_limits<long long>::max();

    // Check all 24 symmetries (6 top colors * 4 rotations each)
    for (colors topColor : { WHITE, YELLOW, RED, ORANGE, GREEN, BLUE }) {
        Cube topChanged = changeTopColor(cube, topColor);
        long long currentNumber = cubeToNumber(topChanged);
        if (currentNumber < minNumber) {
            minNumber = currentNumber;
        }
        for (int i = 0; i < 3; ++i) {
            topChanged = rotate_without_change(topChanged);
            long long currentNumber = cubeToNumber(topChanged);
            if (currentNumber < minNumber) {
                minNumber = currentNumber;
```

```
            }
        }
    }
    return minNumber;
}

// Function to convert enum color to string
std::string colorToString(colors color) {
    switch (color) {
    case WHITE:
        return "WHITE";
    case YELLOW:
        return "YELLOW";
    case RED:
        return "RED";
    case ORANGE:
        return "ORANGE";
    case GREEN:
        return "GREEN";
    case BLUE:
        return "BLUE";
    default:
        return " ";
    }
}

void displayCube(std::ostream& outputFile, const Cube& cube) {
    const std::array<std::string, 8> cornerLabels
        = { "UFR", "UBR", "UBL", "UFL", "DFR", "DBR", "DBL", "DFL" };

    // Write the cube's state to the open file stream
    for (size_t i = 0; i < cube.size(); ++i) {
        outputFile << "Corner { ";
        for (size_t j = 0; j < 3; ++j) {
            outputFile << colorToString(cube[i][j]);
            if (j < 2) outputFile << ", ";
        }
        outputFile << " }, // Corner " << i << ": "
             << cornerLabels[i] << std::endl;
    }
    outputFile << std::endl;
}

// Function to explore all possible configurations using BFS
void findGodsNumber(Cube solvedCube, std::ostream& out
    = std::cout, bool print_example = false, int sample = 1) {
    std::unordered_set<long long> visited;    // To store configurations
    std::queue<std::pair<Cube, int>> queue;   // Queue for BFS

    long long solvedNumber = min_symmetry(solvedCube);
```

```
    queue.push({ solvedCube, 0 });
    visited.insert(solvedNumber);

    int depth = 0;
    std::vector<Cube> deepest_sample;
    while (!queue.empty()) {
        int currentDepth = depth;
        size_t levelSize = queue.size();
        out << "Exploring depth: " << depth << " with "
            << levelSize << " nodes." << std::endl;
        out << "Finding unique states: " << visited.size() << std::endl;
        bool next_level_exist = false;
        for (size_t i = 0; i < levelSize; ++i) {
            // Use explicit access to elements of the pair
            Cube currentCube = queue.front().first;
            queue.pop();

            // Generate possible moves (rotations)
            std::vector<Cube> nextMoves = { rotate_Front(currentCube),
                                            rotate_Front_Inverse(currentCube),
                                            rotate_Right(currentCube),
                                            rotate_Right_Inverse(currentCube),
                                            rotate_Back(currentCube),
                                            rotate_Back_Inverse(currentCube),
                                            rotate_Left(currentCube),
                                            rotate_Left_Inverse(currentCube),
                                            rotate_Top(currentCube),
                                            rotate_Top_Inverse(currentCube),
                                            rotate_Down(currentCube),
                                            rotate_Down_Inverse(currentCube) };
            for (const auto& nextCube : nextMoves) {
                long long nextNumber = min_symmetry(nextCube);
                if (visited.find(nextNumber) == visited.end()) {
                    visited.insert(nextNumber);
                    queue.push({ nextCube, currentDepth + 1 });
                    if (print_example) {
                        if (!next_level_exist) {
                            deepest_sample.clear();
                        }
                        if (deepest_sample.size() < sample) {
                            deepest_sample.push_back(nextCube);
                        }
                        next_level_exist = true;
                    }
                }
            }
        }
        out << "All unique configurations reached at depth: "
            << depth << std::endl << std::endl;
        depth++;
```

```
    }

    out << "Explored all configurations." << std::endl;
    if (print_example) {
        out << "Cubes can be reached at depth "
            << (depth - 1) << " are: " << std::endl;
        for (int i = 0; i < deepest_sample.size(); i++) {
            displayCube(out, deepest_sample[i]);
        }
    }
}


int main() {
    // Initialize a solved cube state with standard corner labels
    std::ofstream outputFile("output_S.txt");

    // Check if the file is open
    if (!outputFile.is_open()) {
        std::cerr << "Failed to open the file." << std::endl;
    }
    Cube cube_6_color = {
        Corner {  WHITE,   BLUE,    RED }, // Corner 0: UFR
        Corner {  WHITE, ORANGE,   BLUE }, // Corner 1: UBR
        Corner {  WHITE,  GREEN, ORANGE }, // Corner 2: UBL
        Corner {  WHITE,    RED,  GREEN }, // Corner 3: UFL
        Corner { YELLOW,    RED,   BLUE }, // Corner 4: DFR
        Corner { YELLOW,   BLUE, ORANGE }, // Corner 5: DBR
        Corner { YELLOW, ORANGE,  GREEN }, // Corner 6: DBL
        Corner { YELLOW,  GREEN,    RED }  // Corner 7: DFL
    };
    Cube cube_2_color_1v5 = {
        Corner {  WHITE, YELLOW, YELLOW }, // Corner 0: UFR
        Corner {  WHITE, YELLOW, YELLOW }, // Corner 1: UBR
        Corner {  WHITE, YELLOW, YELLOW }, // Corner 2: UBL
        Corner {  WHITE, YELLOW, YELLOW }, // Corner 3: UFL
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 4: DFR
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 5: DBR
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 6: DBL
        Corner { YELLOW, YELLOW, YELLOW }  // Corner 7: DFL
    };

    Cube cube_2_color_2v4_symmetric = {
        Corner { WHITE, YELLOW, YELLOW }, // Corner 0: UFR
        Corner { WHITE, YELLOW, YELLOW }, // Corner 1: UBR
        Corner { WHITE, YELLOW, YELLOW }, // Corner 2: UBL
        Corner { WHITE, YELLOW, YELLOW }, // Corner 3: UFL
        Corner { WHITE, YELLOW, YELLOW }, // Corner 4: DFR
        Corner { WHITE, YELLOW, YELLOW }, // Corner 5: DBR
        Corner { WHITE, YELLOW, YELLOW }, // Corner 6: DBL
```

```
        Corner { WHITE, YELLOW, YELLOW }  // Corner 7: DFL
    };
    Cube cube_2_color_2v4_adjacent = {
        Corner {  WHITE,  WHITE, YELLOW }, // Corner 0: UFR
        Corner {  WHITE, YELLOW,  WHITE }, // Corner 1: UBR
        Corner {  WHITE, YELLOW, YELLOW }, // Corner 2: UBL
        Corner {  WHITE, YELLOW, YELLOW }, // Corner 3: UFL
        Corner { YELLOW, YELLOW,  WHITE }, // Corner 4: DFR
        Corner { YELLOW,  WHITE, YELLOW }, // Corner 5: DBR
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 6: DBL
        Corner { YELLOW, YELLOW, YELLOW }  // Corner 7: DFL
    };

    Cube cube_2_color_3v3_all_adjacent = {
        Corner {  WHITE,  WHITE,  WHITE }, // Corner 0: UFR
        Corner {  WHITE, YELLOW,  WHITE }, // Corner 1: UBR
        Corner {  WHITE, YELLOW, YELLOW }, // Corner 2: UBL
        Corner {  WHITE,  WHITE, YELLOW }, // Corner 3: UFL
        Corner { YELLOW,  WHITE,  WHITE }, // Corner 4: DFR
        Corner { YELLOW,  WHITE, YELLOW }, // Corner 5: DBR
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 6: DBL
        Corner { YELLOW, YELLOW,  WHITE }  // Corner 7: DFL
    };

    Cube cube_2_color_3v3_opposite = {
        Corner {  WHITE, YELLOW,  WHITE }, // Corner 0: UFR
        Corner {  WHITE,  WHITE, YELLOW }, // Corner 1: UBR
        Corner {  WHITE, YELLOW,  WHITE }, // Corner 2: UBL
        Corner {  WHITE,  WHITE, YELLOW }, // Corner 3: UFL
        Corner { YELLOW,  WHITE, YELLOW }, // Corner 4: DFR
        Corner { YELLOW, YELLOW,  WHITE }, // Corner 5: DBR
        Corner { YELLOW,  WHITE, YELLOW }, // Corner 6: DBL
        Corner { YELLOW, YELLOW,  WHITE }  // Corner 7: DFL
    };
    outputFile << std::endl << "1v5 Cube:" << std::endl;
    findGodsNumber(cube_2_color_1v5, outputFile);
    outputFile << std::endl << "2v4 Adjacent Cube:" << std::endl;
    findGodsNumber(cube_2_color_2v4_adjacent, outputFile);
    outputFile << std::endl << "2v4 Symmetric Cube:" << std::endl;
    findGodsNumber(cube_2_color_2v4_symmetric, outputFile);
    outputFile << std::endl << "3v3 Opposite Cube:" << std::endl;
    findGodsNumber(cube_2_color_3v3_opposite, outputFile);
    outputFile << std::endl << "3v3 All Adjacent:" << std::endl;
    findGodsNumber(cube_2_color_3v3_all_adjacent, outputFile, true);

    return 0;
}
```

A.2. Codes of Order 3 Cube

For $3 \times 3 \times 3$ Cube, the algorithm, using similar approach for $2 \times 2 \times 2$ Cube, is also attached below:

```
#include <algorithm>
#include <array>
#include <fstream>
#include <iostream>
#include <queue>
#include <unordered_set>
#include <vector>

// define 6 colors in total
enum colors { WHITE = 0, YELLOW = 1, RED = 2,
              ORANGE = 3, GREEN = 4, BLUE = 5 };

// define the structure of corners and 2 by 2 by 2 cube

// The corners are labled as 0 at UFR, 1 at UBR, 2 at UBL,
// 3 at UFL, 4 at DFR, 5 at DBR, 6 at DBL, 7 at DFL
using Corner = std::array<colors, 3>;
using Edge = std::array<colors, 2>;
struct Cube {
    std::array<Corner, 8> corners;   // Array of 8 corners
    std::array<Edge, 12> edges;      // Array of 12 edges
};

// define Front Rotation
Cube rotate_Front(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube.corners[0];
    Corner temp3 = cube.corners[3];
    Corner temp4 = cube.corners[4];
    Corner temp7 = cube.corners[7];
    new_state.corners[0] = { temp3[2], temp3[0], temp3[1] };
    new_state.corners[3] = { temp7[1], temp7[2], temp7[0] };
    new_state.corners[4] = { temp0[1], temp0[2], temp0[0] };
    new_state.corners[7] = { temp4[2], temp4[0], temp4[1] };
    Edge e_temp0 = cube.edges[0];
    Edge e_temp4 = cube.edges[4];
    Edge e_temp8 = cube.edges[8];
    Edge e_temp7 = cube.edges[7];
    new_state.edges[0] = { e_temp7[1], e_temp7[0] };
    new_state.edges[4] = { e_temp0[0], e_temp0[1] };
    new_state.edges[8] = { e_temp4[1], e_temp4[0] };
    new_state.edges[7] = { e_temp8[0], e_temp8[1] };
    return new_state;
}
Cube rotate_Front_Inverse(Cube cube) {
    Cube new_state = cube;
```

```
    Corner temp0 = cube.corners[0];
    Corner temp3 = cube.corners[3];
    Corner temp4 = cube.corners[4];
    Corner temp7 = cube.corners[7];
    new_state.corners[0] = { temp4[2], temp4[0], temp4[1] };
    new_state.corners[3] = { temp0[1], temp0[2], temp0[0] };
    new_state.corners[4] = { temp7[1], temp7[2], temp7[0] };
    new_state.corners[7] = { temp3[2], temp3[0], temp3[1] };
    Edge e_temp0 = cube.edges[0];
    Edge e_temp4 = cube.edges[4];
    Edge e_temp8 = cube.edges[8];
    Edge e_temp7 = cube.edges[7];
    new_state.edges[0] = { e_temp4[0], e_temp4[1] };
    new_state.edges[4] = { e_temp8[1], e_temp8[0] };
    new_state.edges[8] = { e_temp7[0], e_temp7[1] };
    new_state.edges[7] = { e_temp0[1], e_temp0[0] };
    return new_state;
}

// define Right Rotation

Cube rotate_Right(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube.corners[0];
    Corner temp1 = cube.corners[1];
    Corner temp4 = cube.corners[4];
    Corner temp5 = cube.corners[5];
    new_state.corners[0] = { temp4[1], temp4[2], temp4[0] };
    new_state.corners[1] = { temp0[2], temp0[0], temp0[1] };
    new_state.corners[4] = { temp5[2], temp5[0], temp5[1] };
    new_state.corners[5] = { temp1[1], temp1[2], temp1[0] };
    Edge e_temp1 = cube.edges[1];
    Edge e_temp4 = cube.edges[4];
    Edge e_temp5 = cube.edges[5];
    Edge e_temp9 = cube.edges[9];
    new_state.edges[1] = { e_temp4[1], e_temp4[0] };
    new_state.edges[4] = { e_temp9[0], e_temp9[1] };
    new_state.edges[5] = { e_temp1[0], e_temp1[1] };
    new_state.edges[9] = { e_temp5[1], e_temp5[0] };
    return new_state;
}

Cube rotate_Right_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube.corners[0];
    Corner temp1 = cube.corners[1];
    Corner temp4 = cube.corners[4];
    Corner temp5 = cube.corners[5];
    new_state.corners[0] = { temp1[1], temp1[2], temp1[0] };
    new_state.corners[1] = { temp5[2], temp5[0], temp5[1] };
```

```
    new_state.corners[4] = { temp0[2], temp0[0], temp0[1] };
    new_state.corners[5] = { temp4[1], temp4[2], temp4[0] };
    Edge e_temp1 = cube.edges[1];
    Edge e_temp4 = cube.edges[4];
    Edge e_temp5 = cube.edges[5];
    Edge e_temp9 = cube.edges[9];
    new_state.edges[1] = { e_temp5[0], e_temp5[1] };
    new_state.edges[4] = { e_temp1[1], e_temp1[0] };
    new_state.edges[5] = { e_temp9[1], e_temp9[0] };
    new_state.edges[9] = { e_temp4[0], e_temp4[1] };
    return new_state;
}

Cube rotate_Back(Cube cube) {
    Cube new_state = cube;
    Corner temp1 = cube.corners[1];
    Corner temp2 = cube.corners[2];
    Corner temp5 = cube.corners[5];
    Corner temp6 = cube.corners[6];
    new_state.corners[1] = { temp5[1], temp5[2], temp5[0] };
    new_state.corners[2] = { temp1[2], temp1[0], temp1[1] };
    new_state.corners[5] = { temp6[2], temp6[0], temp6[1] };
    new_state.corners[6] = { temp2[1], temp2[2], temp2[0] };
    Edge e_temp2 = cube.edges[2];
    Edge e_temp5 = cube.edges[5];
    Edge e_temp6 = cube.edges[6];
    Edge e_temp10 = cube.edges[10];
    new_state.edges[2] = { e_temp5[1], e_temp5[0] };
    new_state.edges[5] = { e_temp10[0], e_temp10[1] };
    new_state.edges[6] = { e_temp2[0], e_temp2[1] };
    new_state.edges[10] = { e_temp6[1], e_temp6[0] };
    return new_state;
}

Cube rotate_Back_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp1 = cube.corners[1];
    Corner temp2 = cube.corners[2];
    Corner temp5 = cube.corners[5];
    Corner temp6 = cube.corners[6];
    new_state.corners[1] = { temp2[1], temp2[2], temp2[0] };
    new_state.corners[2] = { temp6[2], temp6[0], temp6[1] };
    new_state.corners[5] = { temp1[2], temp1[0], temp1[1] };
    new_state.corners[6] = { temp5[1], temp5[2], temp5[0] };
    Edge e_temp2 = cube.edges[2];
    Edge e_temp5 = cube.edges[5];
    Edge e_temp6 = cube.edges[6];
    Edge e_temp10 = cube.edges[10];
    new_state.edges[2] = { e_temp6[0], e_temp6[1] };
    new_state.edges[5] = { e_temp2[1], e_temp2[0] };
```

```
    new_state.edges[6] = { e_temp10[1], e_temp10[0] };
    new_state.edges[10] = { e_temp5[0], e_temp5[1] };
    return new_state;
}

Cube rotate_Left(Cube cube) {
    Cube new_state = cube;
    Corner temp2 = cube.corners[2];
    Corner temp3 = cube.corners[3];
    Corner temp6 = cube.corners[6];
    Corner temp7 = cube.corners[7];
    new_state.corners[2] = { temp6[1], temp6[2], temp6[0] };
    new_state.corners[3] = { temp2[2], temp2[0], temp2[1] };
    new_state.corners[6] = { temp7[2], temp7[0], temp7[1] };
    new_state.corners[7] = { temp3[1], temp3[2], temp3[0] };
    Edge e_temp3 = cube.edges[3];
    Edge e_temp6 = cube.edges[6];
    Edge e_temp7 = cube.edges[7];
    Edge e_temp11 = cube.edges[11];
    new_state.edges[3] = { e_temp6[1], e_temp6[0] };
    new_state.edges[6] = { e_temp11[0], e_temp11[1] };
    new_state.edges[7] = { e_temp3[0], e_temp3[1] };
    new_state.edges[11] = { e_temp7[1], e_temp7[0] };
    return new_state;
}

Cube rotate_Left_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp2 = cube.corners[2];
    Corner temp3 = cube.corners[3];
    Corner temp6 = cube.corners[6];
    Corner temp7 = cube.corners[7];
    new_state.corners[2] = { temp3[1], temp3[2], temp3[0] };
    new_state.corners[3] = { temp7[2], temp7[0], temp7[1] };
    new_state.corners[6] = { temp2[2], temp2[0], temp2[1] };
    new_state.corners[7] = { temp6[1], temp6[2], temp6[0] };
    Edge e_temp3 = cube.edges[3];
    Edge e_temp6 = cube.edges[6];
    Edge e_temp7 = cube.edges[7];
    Edge e_temp11 = cube.edges[11];
    new_state.edges[3] = { e_temp7[0], e_temp7[1] };
    new_state.edges[6] = { e_temp3[1], e_temp3[0] };
    new_state.edges[7] = { e_temp11[1], e_temp11[0] };
    new_state.edges[11] = { e_temp6[0], e_temp6[1] };
    return new_state;
}

Cube rotate_Top(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube.corners[0];
```

```
    Corner temp1 = cube.corners[1];
    Corner temp2 = cube.corners[2];
    Corner temp3 = cube.corners[3];
    new_state.corners[0] = { temp3[0], temp3[1], temp3[2] };
    new_state.corners[1] = { temp0[0], temp0[1], temp0[2] };
    new_state.corners[2] = { temp1[0], temp1[1], temp1[2] };
    new_state.corners[3] = { temp2[0], temp2[1], temp2[2] };
    Edge e_temp0 = cube.edges[0];
    Edge e_temp1 = cube.edges[1];
    Edge e_temp2 = cube.edges[2];
    Edge e_temp3 = cube.edges[3];
    new_state.edges[0] = { e_temp3[0], e_temp3[1] };
    new_state.edges[1] = { e_temp0[0], e_temp0[1] };
    new_state.edges[2] = { e_temp1[0], e_temp1[1] };
    new_state.edges[3] = { e_temp2[0], e_temp2[1] };
    return new_state;
}

Cube rotate_Top_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp0 = cube.corners[0];
    Corner temp1 = cube.corners[1];
    Corner temp2 = cube.corners[2];
    Corner temp3 = cube.corners[3];
    new_state.corners[0] = { temp1[0], temp1[1], temp1[2] };
    new_state.corners[1] = { temp2[0], temp2[1], temp2[2] };
    new_state.corners[2] = { temp3[0], temp3[1], temp3[2] };
    new_state.corners[3] = { temp0[0], temp0[1], temp0[2] };
    Edge e_temp0 = cube.edges[0];
    Edge e_temp1 = cube.edges[1];
    Edge e_temp2 = cube.edges[2];
    Edge e_temp3 = cube.edges[3];
    new_state.edges[0] = { e_temp1[0], e_temp1[1] };
    new_state.edges[1] = { e_temp2[0], e_temp2[1] };
    new_state.edges[2] = { e_temp3[0], e_temp3[1] };
    new_state.edges[3] = { e_temp0[0], e_temp0[1] };
    return new_state;
}

Cube rotate_Down(Cube cube) {
    Cube new_state = cube;
    Corner temp4 = cube.corners[4];
    Corner temp5 = cube.corners[5];
    Corner temp6 = cube.corners[6];
    Corner temp7 = cube.corners[7];
    new_state.corners[4] = { temp7[0], temp7[1], temp7[2] };
    new_state.corners[5] = { temp4[0], temp4[1], temp4[2] };
    new_state.corners[6] = { temp5[0], temp5[1], temp5[2] };
    new_state.corners[7] = { temp6[0], temp6[1], temp6[2] };
    Edge e_temp8 = cube.edges[8];
```

```
    Edge e_temp9 = cube.edges[9];
    Edge e_temp10 = cube.edges[10];
    Edge e_temp11 = cube.edges[11];
    new_state.edges[8] = { e_temp11[0], e_temp11[1] };
    new_state.edges[9] = { e_temp8[0], e_temp8[1] };
    new_state.edges[10] = { e_temp9[0], e_temp9[1] };
    new_state.edges[11] = { e_temp10[0], e_temp10[1] };
    return new_state;
}

Cube rotate_Down_Inverse(Cube cube) {
    Cube new_state = cube;
    Corner temp4 = cube.corners[4];
    Corner temp5 = cube.corners[5];
    Corner temp6 = cube.corners[6];
    Corner temp7 = cube.corners[7];
    new_state.corners[4] = { temp5[0], temp5[1], temp5[2] };
    new_state.corners[5] = { temp6[0], temp6[1], temp6[2] };
    new_state.corners[6] = { temp7[0], temp7[1], temp7[2] };
    new_state.corners[7] = { temp4[0], temp4[1], temp4[2] };
    Edge e_temp8 = cube.edges[8];
    Edge e_temp9 = cube.edges[9];
    Edge e_temp10 = cube.edges[10];
    Edge e_temp11 = cube.edges[11];
    new_state.edges[8] = { e_temp9[0], e_temp9[1] };
    new_state.edges[9] = { e_temp10[0], e_temp10[1] };
    new_state.edges[10] = { e_temp11[0], e_temp11[1] };
    new_state.edges[11] = { e_temp8[0], e_temp8[1] };
    return new_state;
}

// Convert the cube to a numerical representation for comparison
long long cubeToNumber(const Cube& cube, long long color_num) {
    long long number = 0;
    for (const auto& corner : cube.corners) {
        for (const auto& color : corner) {
            number = number * color_num + color;
        }
    }
    for (const auto& edge : cube.edges) {
        for (const auto& color : edge) {
            number = number * color_num + color;
        }
    }
    return number;
}

// Convert the cube to a numerical representation for comparison
Cube numberToCube(long long cube_num, long long color_num) {
    Cube cube;
```

```cpp
    // Decode edges
    for (int i = cube.edges.size() - 1; i >= 0; --i) {
        for (int j = 1; j >= 0; --j) {
            cube.edges[i][j] = static_cast<colors>(cube_num % color_num);
            cube_num /= color_num;
        }
    }

    // Decode corners
    for (int i = cube.corners.size() - 1; i >= 0; --i) {
        for (int j = 2; j >= 0; --j) {
            cube.corners[i][j] = static_cast<colors>(cube_num % color_num);
            cube_num /= color_num;
        }
    }

    return cube;
}


// Function to convert enum color to string
std::string colorToString(colors color) {
    switch (color) {
    case WHITE:
        return "WHITE";
    case YELLOW:
        return "YELLOW";
    case RED:
        return "RED";
    case ORANGE:
        return "ORANGE";
    case GREEN:
        return "GREEN";
    case BLUE:
        return "BLUE";
    default:
        return " ";
    }
}

void displayCube(std::ostream& outputFile, const Cube& cube) {
    const std::array<std::string, 8> cornerLabels = { "UFR", "UBR", "UBL",
         "UFL", "DFR", "DBR", "DBL", "DFL" };
    const std::array<std::string, 12> edgeLabels
      = { "UF", "UR", "UB", "UL", "FR", "RB",
         "BL", "LF", "DF", "DR", "DB", "DL" };

    // Write the cube's state to the open file stream
    for (size_t i = 0; i < cube.corners.size(); ++i) {
        outputFile << "Corner { ";
```

```
        for (size_t j = 0; j < 3; ++j) {
            outputFile << colorToString(cube.corners[i][j]);
            if (j < 2) outputFile << ", ";
        }
        outputFile << " }, // Corner " << i << ": "
            << cornerLabels[i] << std::endl;
    }
    for (size_t i = 0; i < cube.edges.size(); ++i) {
        outputFile << "Edge { ";
        for (size_t j = 0; j < 2; ++j) {
            outputFile << colorToString(cube.edges[i][j]);
            if (j < 1) outputFile << ", ";
        }
        outputFile << " }, // Edge " << i
            << ": " << edgeLabels[i] << std::endl;
    }
    outputFile << std::endl;
}

// Function to explore all possible configurations using BFS
void findGodsNumber(Cube solvedCube, std::ostream& out = std::cout,
                    bool print_example = false, int sample = 1,
                    long long num_color = 2) {
    std::unordered_set<long long> visited;       // To store configurations
    std::queue<std::pair<long long, int>> queue;   // Queue for BFS

    long long solvedNumber = cubeToNumber(solvedCube, num_color);
    queue.push({ solvedNumber, 0 });
    visited.insert(solvedNumber);

    int depth = 0;
    std::vector<Cube> deepest_sample;
    while (!queue.empty()) {
        int currentDepth = depth;
        size_t levelSize = queue.size();
        out << "Exploring depth: " << depth << " with "
            << levelSize << " nodes." << std::endl;
        out << "Finding unique states: " << visited.size() << std::endl;
        bool next_level_exist = false;
        for (size_t i = 0; i < levelSize; ++i) {
            // Use explicit access to elements of the pair
            long long cube_num = queue.front().first;
            Cube currentCube = numberToCube(cube_num, num_color);
            queue.pop();

            // Generate possible moves (rotations)
            std::vector<Cube> nextMoves = { rotate_Front(currentCube),
                                            rotate_Front_Inverse(currentCube),
                                            rotate_Right(currentCube),
                                            rotate_Right_Inverse(currentCube),
```

```
                                        rotate_Back(currentCube),
                                        rotate_Back_Inverse(currentCube),
                                        rotate_Left(currentCube),
                                        rotate_Left_Inverse(currentCube),
                                        rotate_Top(currentCube),
                                        rotate_Top_Inverse(currentCube),
                                        rotate_Down(currentCube),
                                        rotate_Down_Inverse(currentCube) };
            for (const auto& nextCube : nextMoves) {
                long long nextNumber = cubeToNumber(nextCube, num_color);
                if (visited.find(nextNumber) == visited.end()) {
                    visited.insert(nextNumber);
                    queue.push({ nextNumber, currentDepth + 1 });
                    if (print_example) {
                        if (!next_level_exist) {
                            deepest_sample.clear();
                        }
                        if (deepest_sample.size() < sample) {
                            deepest_sample.push_back(nextCube);
                        }
                        next_level_exist = true;
                    }
                }
            }
        }
        out << "All unique configurations reached at depth: "
            << depth << std::endl << std::endl;
        depth++;
    }

    out << "Explored all configurations." << std::endl;
    if (print_example) {
        out << "Cubes can be reached at depth "
            << (depth - 1) << " are: " << std::endl;
        for (int i = 0; i < deepest_sample.size(); i++) {
            displayCube(out, deepest_sample[i]);
        }
    }
}


int main() {
    // Initialize a solved cube state with standard corner labels
    std::ofstream outputFile("output3.txt");

    // Check if the file is open
    if (!outputFile.is_open()) {
        std::cerr << "Failed to open the file." << std::endl;
    }
    Cube cube_6_color = {
```

```
        Corner { WHITE, BLUE, RED }, // Corner 0: UFR
        Corner { WHITE, ORANGE, BLUE }, // Corner 1: UBR
        Corner { WHITE, GREEN, ORANGE }, // Corner 2: UBL
        Corner { WHITE, RED, GREEN }, // Corner 3: UFL
        Corner { YELLOW, RED, BLUE }, // Corner 4: DFR
        Corner { YELLOW, BLUE, ORANGE }, // Corner 5: DBR
        Corner { YELLOW, ORANGE, GREEN }, // Corner 6: DBL
        Corner { YELLOW, GREEN, RED }, // Corner 7: DFL
        Edge { RED, WHITE }, //  Edge 0: UF
        Edge { BLUE, WHITE }, // Edge 1: UR
        Edge { ORANGE, WHITE }, // Edge 2: UB
        Edge { GREEN, WHITE }, // Edge 3: UL
        Edge { RED, BLUE }, // Edge 4: FR
        Edge { BLUE, ORANGE }, // Edge 5: RB
        Edge { ORANGE, GREEN }, // Edge 6: BL
        Edge { GREEN, RED }, // Edge 7: LF
        Edge { YELLOW, RED }, // Edge 8: DF
        Edge { YELLOW, BLUE }, // Edge 9: DR
        Edge { YELLOW, ORANGE }, // Edge 10: DB
        Edge { YELLOW, GREEN }  // Edge 11: DL
    };
    Cube cube_2_color_1v5 = {
        Corner { WHITE, YELLOW, YELLOW }, // Corner 0: UFR
        Corner { WHITE, YELLOW, YELLOW }, // Corner 1: UBR
        Corner { WHITE, YELLOW, YELLOW }, // Corner 2: UBL
        Corner { WHITE, YELLOW, YELLOW }, // Corner 3: UFL
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 4: DFR
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 5: DBR
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 6: DBL
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 7: DFL
        Edge { YELLOW, WHITE },
        Edge { YELLOW, WHITE },
        Edge { YELLOW, WHITE },
        Edge { YELLOW, WHITE },
        Edge { YELLOW, YELLOW },
        Edge { YELLOW, YELLOW },
        Edge { YELLOW, YELLOW },
        Edge { YELLOW, YELLOW },
        Edge { YELLOW, YELLOW },
        Edge { YELLOW, YELLOW },
        Edge { YELLOW, YELLOW },
        Edge { YELLOW, YELLOW },
    };
    Cube cube_2_color_2v4a = {
        Corner { YELLOW, WHITE, WHITE }, // Corner 0: UFR
        Corner { YELLOW, WHITE, WHITE }, // Corner 1: UBR
        Corner { YELLOW, WHITE, WHITE }, // Corner 2: UBL
        Corner { YELLOW, WHITE, WHITE }, // Corner 3: UFL
        Corner { YELLOW, WHITE, WHITE }, // Corner 4: DFR
        Corner { YELLOW, WHITE, WHITE }, // Corner 5: DBR
```

```
        Corner { YELLOW, WHITE, WHITE }, // Corner 6: DBL
        Corner { YELLOW, WHITE, WHITE }, // Corner 7: DFL
        Edge { WHITE, YELLOW }, //  Edge 0: UF
        Edge { WHITE, YELLOW }, // Edge 1: UR
        Edge { WHITE, YELLOW }, // Edge 2: UB
        Edge { WHITE, YELLOW }, // Edge 3: UL
        Edge { WHITE, WHITE }, // Edge 4: FR
        Edge { WHITE, WHITE }, // Edge 5: RB
        Edge { WHITE, WHITE }, // Edge 6: BL
        Edge { WHITE, WHITE }, // Edge 7: LF
        Edge { YELLOW, WHITE }, // Edge 8: DF
        Edge { YELLOW, WHITE }, // Edge 9: DR
        Edge { YELLOW, WHITE }, // Edge 10: DB
        Edge { YELLOW, WHITE }  // Edge 11: DL
    };
    Cube cube_2_color_2v4b = {
        Corner { WHITE, YELLOW, WHITE }, // Corner 0: UFR
        Corner { WHITE, WHITE, YELLOW }, // Corner 1: UBR
        Corner { WHITE, WHITE, WHITE }, // Corner 2: UBL
        Corner { WHITE, WHITE, WHITE }, // Corner 3: UFL
        Corner { YELLOW, WHITE, YELLOW }, // Corner 4: DFR
        Corner { YELLOW, YELLOW, WHITE }, // Corner 5: DBR
        Corner { YELLOW, WHITE, WHITE }, // Corner 6: DBL
        Corner { YELLOW, WHITE, WHITE }, // Corner 7: DFL
        Edge { WHITE, WHITE }, //  Edge 0: UF
        Edge { YELLOW, WHITE }, // Edge 1: UR
        Edge { WHITE, WHITE }, // Edge 2: UB
        Edge { WHITE, WHITE }, // Edge 3: UL
        Edge { WHITE, YELLOW }, // Edge 4: FR
        Edge { YELLOW, WHITE }, // Edge 5: RB
        Edge { WHITE, WHITE }, // Edge 6: BL
        Edge { WHITE, WHITE }, // Edge 7: LF
        Edge { YELLOW, WHITE }, // Edge 8: DF
        Edge { YELLOW, YELLOW }, // Edge 9: DR
        Edge { YELLOW, WHITE }, // Edge 10: DB
        Edge { YELLOW, WHITE }  // Edge 11: DL
    };
    Cube cube_2_color_3v3a = {
        Corner { YELLOW, YELLOW, WHITE }, // Corner 0: UFR
        Corner { YELLOW, WHITE, YELLOW }, // Corner 1: UBR
        Corner { YELLOW, WHITE, WHITE }, // Corner 2: UBL
        Corner { YELLOW, WHITE, WHITE }, // Corner 3: UFL
        Corner { YELLOW, WHITE, YELLOW }, // Corner 4: DFR
        Corner { YELLOW, YELLOW, WHITE }, // Corner 5: DBR
        Corner { YELLOW, WHITE, WHITE }, // Corner 6: DBL
        Corner { YELLOW, WHITE, WHITE }, // Corner 7: DFL
        Edge { WHITE, YELLOW }, //  Edge 0: UF
        Edge { YELLOW, YELLOW }, // Edge 1: UR
        Edge { WHITE, YELLOW }, // Edge 2: UB
        Edge { WHITE, YELLOW }, // Edge 3: UL
```

```
        Edge { WHITE, YELLOW }, // Edge 4: FR
        Edge { YELLOW, WHITE }, // Edge 5: RB
        Edge { WHITE, WHITE }, // Edge 6: BL
        Edge { WHITE, WHITE }, // Edge 7: LF
        Edge { YELLOW, WHITE }, // Edge 8: DF
        Edge { YELLOW, YELLOW }, // Edge 9: DR
        Edge { YELLOW, WHITE }, // Edge 10: DB
        Edge { YELLOW, WHITE }  // Edge 11: DL
    };
    Cube cube_2_color_3v3b = {
        Corner { WHITE, WHITE, WHITE }, // Corner 0: UFR
        Corner { WHITE, YELLOW, WHITE }, // Corner 1: UBR
        Corner { WHITE, YELLOW, YELLOW }, // Corner 2: UBL
        Corner { WHITE, WHITE, YELLOW }, // Corner 3: UFL
        Corner { YELLOW, WHITE, WHITE }, // Corner 4: DFR
        Corner { YELLOW, WHITE, YELLOW }, // Corner 5: DBR
        Corner { YELLOW, YELLOW, YELLOW }, // Corner 6: DBL
        Corner { YELLOW, YELLOW, WHITE }, // Corner 7: DFL
        Edge { WHITE, WHITE }, //  Edge 0: UF
        Edge { WHITE, WHITE }, // Edge 1: UR
        Edge { YELLOW, WHITE }, // Edge 2: UB
        Edge { YELLOW, WHITE }, // Edge 3: UL
        Edge { WHITE, WHITE }, // Edge 4: FR
        Edge { WHITE, YELLOW }, // Edge 5: RB
        Edge { YELLOW, YELLOW }, // Edge 6: BL
        Edge { YELLOW, WHITE }, // Edge 7: LF
        Edge { YELLOW, WHITE }, // Edge 8: DF
        Edge { YELLOW, WHITE }, // Edge 9: DR
        Edge { YELLOW, YELLOW }, // Edge 10: DB
        Edge { YELLOW, YELLOW }  // Edge 11: DL
    };
    outputFile << std::endl << "2 color 1v5 cube:" << std::endl;
    findGodsNumber(cube_2_color_1v5, outputFile);
    outputFile << std::endl << "2 color 2v4_opposite cube:" << std::endl;
    findGodsNumber(cube_2_color_2v4a, outputFile);
    outputFile << std::endl << "2 color 2v4_adjacent cube:" << std::endl;
    findGodsNumber(cube_2_color_2v4b, outputFile);
    outputFile << std::endl << "2 color 3v3_opposite cube:" << std::endl;
    findGodsNumber(cube_2_color_3v3a, outputFile);
    outputFile << std::endl << "2 color 3v3_adjacent cube:" << std::endl;
    findGodsNumber(cube_2_color_3v3b, outputFile);
    return 0;
}
```

Email address: sjm1@williams.edu, Steven.Miller.MC.96@aya.yale.edu

Department of Mathematics, Williams College, MA 01267

Email address: mtphaovibul@gmail.com

AwesomeMath, White Salmon, WA 98672

Email address: mutian@umich.edu, mtshen1226@gmail.com

University of Michigan, Ann Arbor, MI 48104