Optimal NxNxN Rubik's Cube

Abeer Alsafran

May 16, 2025

1 Introduction

The Rubik's Cube is a 3D combination puzzle invented in 1974 by Hungarian sculptor and professor of architecture Ernő Rubik. Originally called the Magic Cube, the puzzle was licensed by Rubik to be sold by Pentangle Puzzles in the UK in 1978, and then by Ideal Toy Corp in 1980 via businessman Tibor Laczi and Seven Towns founder Tom Kremer. The cube was released internationally in 1980 and became one of the most recognized icons in popular culture. It won the 1980 German Game of the Year special award for Best Puzzle. As of January 2024, around 500 million cubes had been sold worldwide, making it the world's bestselling puzzle game and bestselling toy. The Rubik's Cube was inducted into the US National Toy Hall of Fame in 2014.[1]

2 Glossary

- Cubie: State it's X,Y and Z.
- Sticker: Specify the face of on which that sticker resides (i,e -Z or bottom), and also the two coordinates of the sticker along the surface of the face (i.e (X,Y) on -Z face).See Figure 2 for more clarification.
- RC: Rubik Cube.

3 NP-Completeness

The NxNxN Rubik's Cube considered as an NP-Complete problem. Proof in section 3.1 show that the NxNxN Rubik's Cube problem is NP-Complete Before we dive into the proof, there are some information that need to be cleared: [2].

- Some algorithms depends on the number of moves, and others depends on the time taken to decide the optimality of the solution.
- The move count metric of the cube has a list, we will focus on the following:

Figure 1: The 3x3x3 Rubik's Cube.

– **STM:** The slice turn metric (STM) is a metric for the 3x3x3 where any turn of any layer, by any angle, counts as one turn. This differs from HTM in that a slice move counts as one turn, not two. And it differs from the ATM in that it does not include anti-slice turns and other axial turns. Cube rotations do not count as turns. STM is a very popular metric for those who use methods with many slice turns.

– **QSTM (also known as SQTM):** short for Quarter Slice Turn Metric, is a move count metric for the 3x3x3 in which any clockwise or counterclockwise 90-degree turn of any layer counts as one turn, and rotations do not count as moves.[3]

3.1 Prove

To prove this, we say that a language S is **NP-Complete**, if $\forall L \in NP$, $L \leq^P S$ and $S \in NP$ This means that for all languages in NP (Nondeterministic Polynomial), there is a language that can be reduced in polynomial time to S (Rubik's Cube) and this S is NP (Nondeterministic Polynomial). NxNxN RC is NP: There exists a verifier V that takes as input: a sequence of scrambled states and a sequence of target states, then verifies it in polynomial time. Since each move is a constant. The following is a formal proof for reduction[4].

3.1.1 Hamiltonian Problems:

1- Grid Graph Hamiltonian Cycle problem (GGHC): The Hamiltonian cycle problem is a special case of the traveling salesman problem, obtained by setting the distance between two cities to one if they are adjacent and two otherwise, and verifying that the total distance traveled is equal to n. If so, the route is a Hamiltonian cycle.[5]

2- Promise Grid Graph Hamiltonian Path Problem (GGHP): a topic discussed in the fields of complexity theory and graph theory. It decides if a directed or undirected graph, G, contains a Hamiltonian path, a path that visits every vertex in the graph exactly once. The problem may specify the start and end of the path, in which case the starting vertex s and ending vertex t must be identified.[5]

3- Promise Cubical Hamiltonian Path Problem (CHP):The cubical graph is the Platonic graph corresponding to the connectivity of the cube. It is isomorphic to the generalized Petersen graph.[6]

3.1.2 The Reduction:

In order to reduce from L to S, we need to first reduce from the Grid Graph Hamiltonian Cycle problem (GGHC) to Promise Grid Graph Hamiltonian Path Problem (GGHP), and then reduce the Grid Graph Hamiltonian Path Problem to Promise Cubical Hamiltonian Path Problem (CHP). Second we need to reduce the Cubical Hamiltonian Path Problem to NxNxN Rubik's Cube Problem (RC).[Refer to section 3.1.1 for more information about these problems]

$$GGHC \leq^P GGHP \leq^P CHP \leq^P RC$$

If n (the size of the cube) is odd, then the coordinates of the cubie is -a,-(a-1), ... ,-1,0,1,...,a-1,a, else, if n is even, then the coordinated of the cubies $-a, -(a-1), ..., -1 \cup 1, ..., a-1, a$ This scheme will help if a move relocates a sticker, the coordinates of the sticker remain the same. We need to distinguish the sets of the cubies affected by a move from each other.

- Each move (STM) affects a single slice of n^2 cubies in the same coordinates.
- For the six slices, we refer to the boundary of the cube as 'face slice' (i.e +X face).

X_i, Y_i, Z_i in RC_n, are the transformations into clockwise turns of X,Y, and Z with index i, where i uses the set of slices in NxNxN RC_n which is generated by the set of group elements.[7]

C_o is a canonical solved configuration of RC. Let C_o have the colors in Table 1, then we can construct a configuration of the corresponding RC by applying that element to C_o. Every transformation $t \in RC_n$, corresponds to the configuration $C_t = t(C_o)$. Apply t to C_o.

Axis	Color
+X	Orange
-X	Red
+Y	Green
-Y	Yellow
+Z	White
-Z	Blue

Table 1: Colors correspond to axis

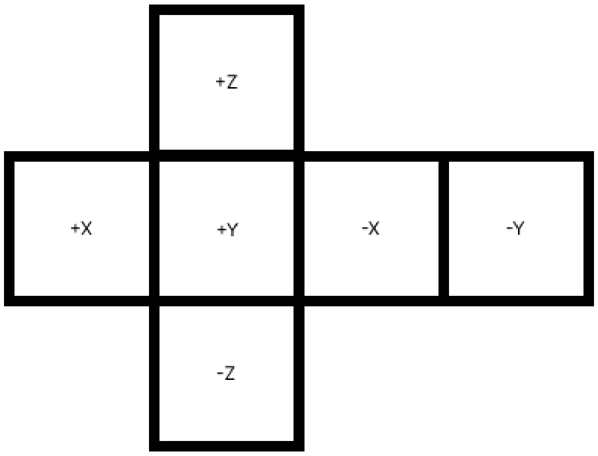

Figure 2: Flattened Cube Grid and Axis

- We want to decide whether the transformation t can be reversed by a sequence of at most k transformations corresponding to legal RC moves under the (STM) metric.
- If (t,k) is a satisfiable instance to the (STM) RC, then (t(C_o ,k)) satisfiable instance to a (STM) RC.

The Cubical Hamiltonian path problem asks if it's possible to re-arrange bit-string s $l_1, ..l_n$ into new order such that each bit-string has hamming distance from the next. It takes as input a cubical graph whose vertices are of length m bit-strings $l_1, .., l_n$. If $l_i, ..., l_n$ is a "yes" instance to the cubical Hamiltonian Path, then (t,k) is a "yes" instance.

To prove that S=NxNxN Rubik's Cube is NP-complete. We need to show that, a language S if $\forall L \in NP, L \leq^P SandS \in NP$ L = CHP. If we proved that CHP can be reduced to Rubik's Cube then we proved that Rubik's Cube is NP_Complete. The b_i terms commute, and so if the input Promise Cubical Hamiltonian Path instance is a "yes" instance then the b_i's can be reordered so that all but k moves in the definition of t will cancel. This case, t can be both enacted and reversed in k moves. rotation of the faces can put rows of stickers that were once aligned parallel to one axis into alignment with another axis. The slices in this construction which will take the role of columns 1 through m.

Suppose we have instance of the CHP consisting of n bit-strings $l_1, .., l_n$ of length m. To construct an RC instance we need to compute the value l indicating the allowed number of moves and construct the transformation t in RC, k = 2n -1. The transformation t will be an element of RC_s where s = 6n+2m, 6 is because we have 6 sides for the cube, each with different colors.

Define a_i for $1 \leq i \leq n$ to be $(x_1)^{(l_i)_1} o...o(x_m)^{(l_i)_m}$ where $(l_i)_1, ..., (l_i)_m$ are the bits of l_i, $b_i = a_1 o b_1 o....o b_n$, the output (t,k) completes the reduction from CHP to RC. (C_t , k) = (t(C_o), k). This reduction is in polynomial time.[7]

4 Survey Algorithms

4.1 Brute-force algorithm

There exist a brute-force algorithm that can solve the NxNxN Rubik's Cube problem by determining a sequence of moves which checks every possible position of the cube, and when it comes to solving it, you will need to follow a sequence **until** you eventually solve it [8], this will take as move for 3x3x3 Rubik's Cube approximately

$$43252003274489856000 \approx 4.3 \cdot 10^{19}$$

To break it down, we have 8 corners in all cubes, the ways to arrange all corners is 8!, but the number of arrangements for the available corners is 8!/2. Each corner can be oriented in 3 states exept for the last corner, so we have 3^7. There are $4*n$ edges on the cube, each edge has 2 colors so, $(n*4)!*2^{(4*n-1)}$[9]. **The Devil's Algorithm** is defined as a set of moves that when applied repeatedly if necessary, will eventually return a Rubik's Cube to a solved state regardless of the starting configuration[10]. It is known that the devil's algorithm is a conceptual theory algorithm that is the inverse of the God's algorithm which solves the RC in the most optimal time. If we take the above mentioned example, regarding the complexity; we can analyze that it takes approximately O($2^{(4*n-1)}$). which is very complex and huge.

4.2 Using heuristics and approximation part 1

There exist another way to solve the NxNxN Rubik's Cube problem efficiently using the IDA^* algorithm[11]. The IDA^* (Iterative Deepening A^*) algorithm is considered a search algorithm that uses heuristics and DFS (Depth First Search) to find the optimal solution for a Rubik's Cube. This means it needs space to store the heuristics dataset of the state of the cubes, so in this case, space complexity matters for achieving optimal solution. The time complexity is not less important than the space complexity, the time complexity in Korf's [12] is O(n/m)[13], where, m is the amount of memory used and n is the size of the problem space. In simple, the algorithm focuses in solving sub-goals like edges and corners, First, the algorithm build the database pattern matching database once at the beginning which store the exact number of moves needed to solve these sub-parts from any possible configuration (to make it faster). Second, using IDA^* it starts a depth-first search but limits how far it goes based on the estimated cost from the current state to the goal (using the heuristics), if it can't find a solution within the current limit, it increases the limit and tries again, until it finds the optimal path. As advantage for this algorithm specifically, it avoid duplicating solutions by not traversing the same path again and it does not store all visited states.

4.3 Using heuristics and approximation part 2

Also, there exist another way to solve the NxNxN Rubik's Cube problem using deep neural network (DNN). In general, it works by approximating the value of the iteration to train a deep neural network to approximate a function that will output the cost to reach the goal. This DNN then combined with the search algorithm A^* that helps approximate and compare the values of different cube states, while searching for the optimal solution. A^* is a popular search algorithm that finds the most smallest path to a given node.[14] It is done by doing the following function: $f(n) = g(n) + h(n)$ where g(n) is the cost of the path from the start node, h(n) is a heuristic that estimates the cost of the cheapest path from n to the goal, where f(n) is the result of g(n) and h(n) combined.
The implemented version [15] of DeepCubeA depends on the training dataset in the DNN. It consists of two stages, the first one is Deep Approximate Value Iteration (DAVI), offline training of a deep neural network (DNN) to estimate the cost-to-go function. Second, Batch Weighted A Search (BWAS)* , a variant of A* search that uses the trained DNN as a heuristic. In their paper, they have showed the result of the DeepCubeA

$$\mathcal{O}(N \cdot T_{\text{DNN}})$$

where:

- N: number of nodes actually expanded
- T_{DNN}: the time to evaluate the deep neural network

Algorithm	Time Complexity
Brute-Force	O($2^{(4*n-1)}$)
IDA^*	O(n/m)
DeepCubeA	O($N \cdot T_{\text{DNN}}$)

Table 2: Time Complexity Comparison

4.4 Time Complexity Comparison of the 3 Survey Algorithms

In this section we compare the time complexity of the 3 different algorithms. In table 2 we can see the difference in time complexity for the 3 survey algorithms, Brute-force, Using heuristics part 1, and part 2. It is obvious that the Brute-force is the most expensive time complexity, because it searches for all the available solutions without any approximation. What makes DeepCubeA special is that it uses Deep Neural Networks to make it self better in every run it makes, giving it the priority for excelling other algorithms such as the one in part 1 (IDA^*), which uses the same idea of pattern database but not the DNN functionality.

5 Proposed Algorithm

Having seen the above algorithms, we now want to propose a new algorithm that performs somewhat better than the brute-force algorithm. We concluded from part 1 and 2 that, to get the best performance for time complexity is by utilizing the pattern database, in which it helps to increase the performance and lower the time complexity for the overall algorithm. Since the Kociemba's algorithm is the most recent algorithm that has the best accuracy so far, we propose to introduce the parallel concept into the Kociemba's, to give it a better performance with respect to time[16].
What **Kociemba's Algorithm** is doing is the following: Phase 1 the Lookup Tables : The code implements several lookup tables for different solving phases. Phase 2 the Centers staging: getting centers of same color together. Phase 3 the Edge pairing: pairing up matching edges. Phase 4 the Reduction: reducing to 3x3x3 (making the 4x4x4 behave like a 3x3x3).

The lookup tables can be parallelized using Python's multiprocessing module, the edge pattern processing can be parallelized as well, and the center stage calculations can be parallelized. We have applied the parallelization on NxNxN Rubik's Cube to test and prove that the parallelization can affect it by increasing the performance and decreasing the time complexity. The modified part is in Appendix A.

Algorithm	Time Complexity
Brute-Force	O($2^{(4*n-1)}$)
IDA^*	O(n/m)
DeepCubeA	O($N \cdot T_{\text{DNN}}$)
Proposed Algorithm	O(N)

Table 3: Proposed Algorithm Time Complexity Comparison

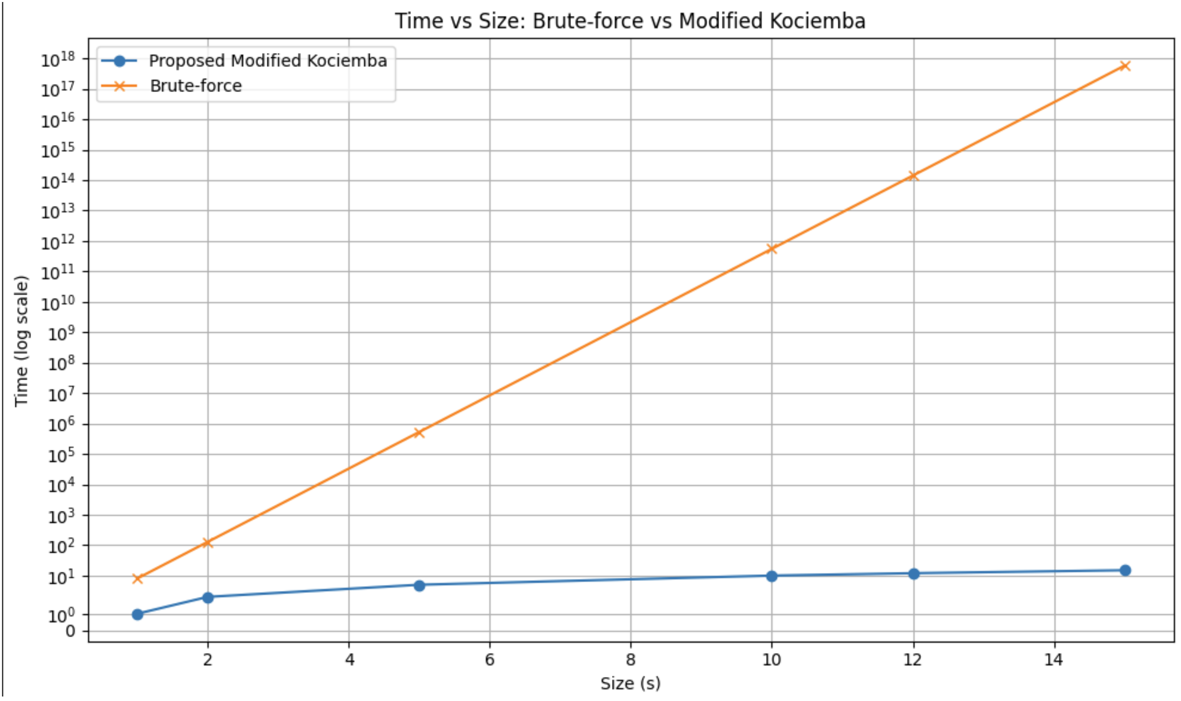

Figure 3: Size and corresponding time

5.1 Compare proposed algorithm with the survey algorithms

We can see that Table 3 shows the comparison between the proposed algorithm and the brute force algorithm. We will visually compare the difference between the brute-force and the proposed algorithm in terms of time and size. As you can see, Figure 3 shows that the proposed algorithm is taking a linear line and brute force is taking an exponential curve. As you can see in Figure 3, size 15 in brute-force takes a very significant time, and size 16 is the limit when it comes to calculating the time.

6 Conclusion

The NxNxN Rubik's Cube is NP-Complete problem (proved). The Cubical Hamiltonian Path Problem is reduced to it. This means that if we can crack any NP problem problem in an efficient (Polynomial-time) way, then we can crack as well all problems that are NP-Complete by applying the reduction and prove its NP. It is not easy to show that the NP problem is P, but with research we may came with a solution that probably can lead to a better chance in proving that NP and P are equivalent.

7 Appendix A

```
from multiprocessing import Pool, cpu_count

# Parallel lookup tables
class LookupTableIDA444Phase3(LookupTableIDAViaGraph):
    def __init__(self, parent):
        # Some code
        self.pool = Pool(processes=cpu_count())

    def parallel_search(self, args):
        move_group, state = args
        results = []
        for move in move_group:
            new_state = self.rotate_state(state, move)
            if self.is_valid_state(new_state):
                results.append((move, new_state))
        return results

    def search(self, state):
        # Split moves into groups for parallel processing
        move_groups = [moves_444[i:i + cpu_count()] for i in range(0,
 len(moves_444), cpu_count())]
        search_args = [(group, state) for group in move_groups]

        # Run parallel searches
        results = self.pool.map(self.parallel_search, search_args)

        # Combine results
        all_results = []
        for group_result in results:
            all_results.extend(group_result)

        return all_results

    def __del__(self):
        if hasattr(self, 'pool'):
            self.pool.close()
            self.pool.join()

# Some code

# Parallelize edge pattern processing
def parallel_process_edge_pattern(args):
    edge_index, square_index, partner_index, state = args
    square_value = state[square_index]
    partner_value = state[partner_index]
    wing_str = wing_str_map[square_value + partner_value]
    return (edge_index, wing_str)

def edges_recolor_pattern_444(state: List[int], only_colors: List[str] = None) -> str:
    edge_map = {
        "UB": [], "UL": [], "UR": [], "UF": [],
        "LB": [], "LF": [], "RB": [], "RF": [],
        "DB": [], "DL": [], "DR": [], "DF": [],
        "--": [],
    }


    with Pool(processes=cpu_count()) as pool:
        args = [(edge_index, square_index, partner_index, state)
for edge_index, square_index, partner_index in
wings_for_edges_recolor_pattern_444]
        results = pool.map(parallel_process_edge_pattern, args)

        for edge_index, wing_str in results:
            edge_map[wing_str].append(edge_index)

    # Some code

# Some Code

# Parallel # Process center groups
class LookupTable444UDCentersStage(LookupTable):
    def __init__(self, parent, build_state_index: bool = False):
        # Some code
        self.pool = Pool(processes=cpu_count())

    def parallel_state_check(self, centers_group):
    return ["U" if self.parent.state[x] in ("U", "D") else "x" for x in centers_group]

    def state(self) -> str:
        # Split centers into groups for parallel processing
    center_groups = [centers_444[i:i + cpu_count()] for i in range(0, len(centers_444),


        results = self.pool.map(self.parallel_state_check, center_groups)

        # Combine results
        return "".join([item for sublist in results for item in sublist])

    def __del__(self):
        if hasattr(self, 'pool'):
            self.pool.close()
            self.pool.join()

# Some code
```

References

[1] Wikipedia, “Rubik’s cube,” *Wikipedia*, 2025.

[2] E. D. Demaine, M. L. Demaine, S. Eisenstat, A. Lubiw, and A. Winslow, “Algorithms for solving rubik’s cubes,” in *Algorithms–ESA 2011: 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings 19*, pp. 689–700, Springer, 2011.

[3] S. Solving, “Metric - speedsolving.com,” *speedsolving*, 2025.

[4] E. D. Demaine, S. Eisenstat, and M. Rudoy, “Solving the Rubik’s Cube Optimally is NP-complete,” in *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)* (R. Niedermeier and B. Vallée, eds.), vol. 96 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 24:1–24:13, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[5] Wikipedia, “Hamiltonian path problem,” *Wekipedia*, 2024.

[6] MathWorld, “Cubical hamiltonian path problem,” *Wolfram Research*, 2025.

[7] E. D. Demaine, S. Eisenstat, and M. Rudoy, “Solving the Rubik’s Cube Optimally is NP-complete,” in *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)* (R. Niedermeier and B. Vallée, eds.), vol. 96 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 24:1–24:13, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[8] Mathematics, “Brute force method of solving the cube: how many moves would it take to solve it?,” *Medium*, 2018.

[9] A. Chilakapati, “Understanding the rubik’s cube and why it has over 43 quintillion permutations!,” *cubelelo*, 2025.

[10] Mike, “The devil’s number and the devils’ algorithm,” *anttila*, 2014.

[11] B. Bellerose, “Rubik’s cube solver,” *Stack Exchange*, 2022.

[12] R. E. Korf, “Finding optimal solutions to rubik’s cube using pattern databases,” in *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI’97/IAAI’97, p. 700–705, AAAI Press, 1997.

[13] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, “Iterative deepening a*,” *Wikipedia*.

[14] H. Kociemba, “Rubikscube-optimalsolver,” *Medium*, 2021.

[15] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, “Solving the rubik’s cube with deep reinforcement learning and search,” *Nature Machine Intelligence*, vol. 1, no. 8, pp. 356–363, 2019.

[16] D. Walton, “rubiks-cube-nxnxn-solver,” *GitHub*, 2023.