A Comprehensive Report on Solving the Rubik's Cube using Search Algorithms

Student Name: Ali Qeblawi

Student ID: 48259794

Email: ali.qeblawi@ue-germany.de

Group: B

Course Title: Machine Learning and Smart Systems

Instructor: Prof. Dr. Raja Hashim Ali

Abstract

This report presents a comprehensive investigation into the application and comparative performance of fundamental search algorithms—Depth-First Search (DFS), Breadth-First Search (BFS), and A* search—for solving the 2x2x2 Rubik's Cube. The project deliberately focuses on the 2x2x2 variant (Pocket Cube) to enable a detailed exploration of algorithmic behaviors and ensure verifiable correctness within a computationally tractable state space. This focused approach addresses the significant challenge of state space explosion, which renders these algorithms impractical for larger N>2 cubes without substantial modifications. For the A* algorithm, a custom heuristic function, termed `sticker_manhattan_heuristic_2x2`, was designed and implemented. This heuristic estimates the distance to the goal state by summing the "face distances" of misplaced stickers and applying a scaling factor to promote admissibility.

Experimental evaluations were conducted by testing the algorithms on a variety of 2x2x2 cube scrambles of different depths. Key performance metrics, including solution path length, number of nodes explored, and execution time, were recorded and analyzed. The results consistently demonstrate that A*, guided by the `sticker_manhattan_heuristic_2x2`, finds optimal solutions with markedly fewer nodes explored and significantly reduced computation time compared to both BFS and DFS. BFS, while guaranteeing optimality, incurs high computational costs due to its exhaustive layer-by-layer exploration. DFS, implemented with a depth limit corresponding to the 2x2x2 "God's Number," offers faster exploration in some scenarios

but does not inherently guarantee optimality if the limit is too restrictive or if paths to the goal exceed it.

The study concludes that for the 2x2x2 Rubik's Cube, the A* search algorithm provides the most effective balance between solution optimality and computational efficiency. However, the report also critically discusses the inherent scalability limitations of these algorithms and the implemented A* heuristic when faced with larger N x N x N cubes. The exponential growth in state space and the diminishing informativeness of simple heuristics necessitate the adoption of more advanced data structures, powerful problem-specific heuristics (like Pattern Databases), and potentially different algorithmic paradigms (e.g., Kociemba's algorithm) for effectively tackling generalized Rubik's Cube solving. This research underscores the foundational trade-offs in algorithm design and the importance of heuristic quality in informed search.

1. Introduction

The Rubik's Cube, invented by Ernő Rubik in 1974, transcends its status as a popular puzzle, standing as a canonical problem in computer science, particularly within the domain of artificial intelligence (AI), algorithm design, and computational group theory. Its intricate structure, characterized by a vast number of possible configurations (states), provides a rich testbed for evaluating the efficacy of various problem-solving strategies, especially search algorithms. The challenge lies in finding a sequence of moves (a path) from a given scrambled state to the unique solved state.

1.1. Project Objective

The primary objective of this project is to implement, rigorously test, and critically analyze the performance of three fundamental search algorithms in solving the Rubik's Cube:

- **Depth-First Search (DFS):** An uninformed search algorithm that explores as far as possible along each branch before backtracking.
- **Breadth-First Search (BFS):** An uninformed search algorithm that explores all neighbor nodes at the present depth prior to moving on to nodes at the next depth level, guaranteeing the shortest path in terms of moves for unweighted graphs.
- **A* Search:** An informed search algorithm that employs a heuristic function to guide its exploration towards the most promising states, aiming to find an optimal solution more efficiently than uninformed methods.

1.2. Focus on the 2x2x2 Cube and Justification

While the ultimate goal in Rubik's Cube research often involves solving the standard 3x3x3 cube or even larger N x N x N variants, this project makes a strategic decision to focus its implementation and primary analysis on the 2x2x2 Rubik's Cube (also known as the Pocket Cube). This choice is underpinned by several critical considerations:

- **State Space Manageability:** The 2x2x2 cube has a significantly smaller state space (exactly 3,674,160 reachable configurations from the solved state) compared to the 3x3x3 cube (approximately 4.3times1019 states) or larger cubes (see Section 5.3.1 for state space details). This smaller state space makes it feasible to:
    - Thoroughly test the correctness of the implemented algorithms.
    - Observe and analyze the complete search process for many scrambles.
    - Evaluate the performance of BFS without it becoming immediately intractable.
- **Algorithmic Fundamentals:** The 2x2x2 cube retains the core challenges of state representation, move generation, and search space traversal inherent in all Rubik's Cube variants. It provides a sufficient platform for understanding the fundamental mechanics and trade-offs of DFS, BFS, and A*.
- **Heuristic Development Context:** It allows for the development and testing of a custom A* heuristic (`sticker_manhattan_heuristic_2x2`) in a controlled environment where its behavior and admissibility can be more easily assessed.
- **Intractability for N>2 with Basic Algorithms:** As explicitly stated in the project's aim, preliminary research and established AI literature indicate that for N>2 cubes, uninformed search algorithms like BFS and DFS face severe intractability due to the "curse of dimensionality" or state space explosion. Even A* with simple, general-purpose heuristics struggles immensely. My initial examination of existing complex solvers (e.g., available on GitHub like `Ciriously/AICUBESOLVER` which often use highly specialized techniques like Kociemba's two-phase algorithm for the 3x3x3 cube) reinforced this understanding.

Therefore, this project focuses on the 2x2x2 cube to rigorously explore the core mechanics of DFS, BFS, and A*, ensure correct implementation, and deeply understand their realistic capabilities and limitations before discussing theoretical extensions and practical challenges for larger N.

1.3. Report Structure

This report is organized as follows:

- **Section 2 (Literature Review):** Discusses existing work on Rubik's Cube solvers, relevant search algorithms, heuristic design, and key theoretical concepts like "God's Number."
- **Section 3 (Methodology):** Details the chosen state representation, implementation of move functions, the scrambling process, and the specific design of the DFS, BFS, and A* algorithms, including a thorough explanation of the A* heuristic.
- **Section 4 (Results):** Presents the experimental setup and empirical results obtained from testing the algorithms, utilizing tables and graphs for clarity.
- **Section 5 (Analysis & Discussion):** Provides a critical analysis of the results, comparing algorithm performance, evaluating heuristic effectiveness, and discussing the crucial aspect of scalability to larger cubes.
- **Section 6 (Conclusion):** Summarizes the key findings, reiterates the project's main conclusions, and suggests potential avenues for future work.
- **Section 7 (References):** Lists all cited academic papers, articles, and relevant online resources.

2. Literature Review

This section reviews existing literature and resources pertinent to solving the Rubik's Cube using computational methods. It covers notable solver projects, the theoretical underpinnings of the search algorithms employed, common heuristic strategies for A*, and the concept of "God's Number."

2.1. Existing Rubik's Cube Solvers and Algorithmic Complexity

The quest to solve the Rubik's Cube computationally has led to a plethora of projects and research.

- **Early Solvers and Brute Force:** Early attempts often highlighted the sheer scale of the problem. Brute-force approaches are generally infeasible beyond trivial cases.
- **Specialized Algorithms:** For the 3x3x3 cube, Herbert Kociemba's two-phase algorithm is renowned. It first searches for a state solvable within a small number of moves using only a restricted move set (e.g., U, D, R2, L2, F2, B2), and then solves the cube from this intermediate state. Projects like `Ciriously/AICUBESOLVER` often implement or adapt such sophisticated, multi-stage algorithms.
- **General Search Approaches:** Other projects focus on applying more general AI search techniques. For instance, the `Gualor/rubik-solver` project on GitHub specifically explores solving the 2x2x2 cube using A* with various heuristics, including misplaced colors and pattern databases, providing a relevant comparison point for this project's focus.

- **Machine Learning Approaches:** More recently, machine learning, including deep reinforcement learning, has been applied to Rubik's Cube solving (e.g., OpenAI's work, McAleer et al., 2019), often learning heuristics or solution strategies directly from experience. While beyond this project's scope, these represent the cutting edge.

These examples illustrate a spectrum from general search to highly specialized, domain-specific solutions, often trading off generality for performance on a particular cube size.

2.2. Application of BFS, DFS, and A* to Pathfinding Puzzles

The Rubik's Cube is fundamentally a pathfinding problem on a graph where states are nodes and moves are edges.

- **Breadth-First Search (BFS):** As established by Moore (1959), BFS is guaranteed to find the shortest path in terms of the number of edges (moves) in an unweighted graph. However, its space complexity, O(bd) (where b is the branching factor and d is the solution depth), makes it impractical for large state spaces like the 3x3x3 cube. A student paper by Ignatius (2022) comparing Branch & Bound with BFS for Rubik's Cube solving reiterated BFS's limitations with increasing scramble depth due to memory and time constraints.
- **Depth-First Search (DFS):** DFS explores one branch of the search tree as deeply as possible. Its space complexity is O(bd) or O(d) if not storing all nodes on the current path. However, standard DFS is not optimal and can get trapped in infinite branches if cycles are not handled or if the search space is infinite. Depth-limited DFS or iterative deepening DFS (IDA*) are common improvements.
- **A* Search:** Developed by Hart, Nilsson, and Raphael (1968), A* is an informed search algorithm that uses a heuristic function h(n) to estimate the cost from a node n to the goal. It prioritizes nodes with the lowest f(n)=g(n)+h(n), where g(n) is the exact cost from the start to n. A* is optimal and complete if h(n) is admissible (i.e., it never overestimates the true cost to the goal) and consistent (monotonic). The efficiency of A* heavily depends on the quality of its heuristic. Korf's seminal work on Iterative Deepening A* (IDA*) (Korf, 1985) provided a memory-efficient way to use A*-like search, particularly relevant for problems like the Rubik's Cube. Recent academic works, such as Czechowski's PhD thesis (2023) on deep learning for planning, often discuss classical search algorithms like A* as baselines or components in more advanced hybrid systems, highlighting their continued relevance.

2.3. Common A* Heuristics for the Rubik's Cube

The quality of the heuristic is paramount for A*.

- **Manhattan Distance (MD):** A common heuristic for grid-based puzzles. For the Rubik's Cube, it's adapted to sum the minimum number of moves required to get each individual cubie (or sometimes sticker) to its solved position and orientation.
    - **Cubie-based MD:** For a 3x3x3 cube, this involves summing the 3D Manhattan distances for the 8 corner cubies and 12 edge cubies separately to their solved locations. The total sum is often divided by a factor (e.g., 4 or 8, depending on how moves are counted and pieces are affected) to maintain admissibility. For a 2x2x2 cube, which only has 8 corner cubies, a similar principle applies.
    - **Sticker-based MD (Simpler Variant):** Counts the number of stickers not on their correct target face color. This is conceptually simpler than full 3D cubie MD but generally less informative. The heuristic developed in this project (`sticker_manhattan_heuristic_2x2`) is a variant of this.
- **Pattern Databases (PDBs):** Introduced by Culberson and Schaeffer (1998), PDBs are lookup tables that store the exact optimal solution cost (number of moves) from any configuration of a subset of pieces (the "pattern") to their solved state within that pattern. For instance, a PDB for a 3x3x3 cube might store the costs for all configurations of the 8 corner pieces, ignoring edges.
    - **Construction:** PDBs are typically built by performing a BFS backwards from the goal state of the pattern.
    - **Usage:** During an A* search, the heuristic value for a state is retrieved from the PDB (or multiple PDBs).
    - **Additive PDBs:** If multiple PDBs for disjoint sets of pieces are created (Korf & Felner, 2002), their heuristic values can be summed to provide a much more powerful (and still admissible) heuristic. PDBs can dramatically reduce the search space but require significant precomputation time and storage. For the 2x2x2 cube, a PDB storing distances for all 3.67 million states would be a perfect heuristic.

2.4. "God's Number"

"God's Number" refers to the diameter of the state space graph of the Rubik's Cube – the maximum number of moves required to solve any scrambled configuration using an optimal (shortest) solution.

- **For the 2x2x2 Cube:** God's Number is 11 in the Half-Turn Metric (HTM, where a 180-degree turn counts as one move) or 14 in the Quarter-Turn Metric (QTM, where a 90-degree turn is one move, and a 180-degree turn is two 90-degree

moves). This project primarily uses the QTM implicitly by defining 12 basic 90-degree moves (U, U', D, D', etc.).
- **For the 3x3x3 Cube:** God's Number was proven to be 20 (HTM) in July 2010 by Rokicki, Kociemba, Davidson, and Dethridge, a significant computational achievement (cube20.org). In QTM, it's 26.

Understanding God's Number provides a crucial benchmark for the maximum length of an optimal solution and can inform the setting of depth limits in algorithms like DFS or IDA*.

3. Methodology

This section outlines the technical approach employed in this project, covering the representation of the Rubik's Cube state, the implementation of cube manipulation functions, the method for generating scrambled cube instances, and the detailed design of the three search algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), and A* search, with particular attention to the custom heuristic developed for A*. All implementations were done in Python 3.

3.1. State Representation

A crucial aspect of solving the Rubik's Cube computationally is defining an effective representation of its state. For this project, the state of the 2x2x2 cube is represented as an immutable tuple. This choice is driven by the need to store states in set data structures (for visited tracking in search algorithms) or use them as keys in dictionaries, both of which require hashable objects.

The specific structure is a main tuple containing six elements, where each element represents a face of the cube. Each face, in turn, is represented as a tuple of two rows, and each row is a tuple of two sticker characters (e.g., 'W' for White, 'R' for Red, 'G' for Green, 'B' for Blue, 'O' for Orange, 'Y' for Yellow).

Example (Conceptual Solved State Structure for a 2x2x2 cube): Assume a canonical order of faces, for example: Up (U), Left (L), Front (F), Right (R), Back (B), Down (D).

```
U: (('W', 'W'), ('W', 'W'))
L: (('O', 'O'), ('O', 'O'))
F: (('G', 'G'), ('G', 'G'))
R: (('R', 'R'), ('R', 'R'))
B: (('B', 'B'), ('B', 'B'))
D: (('Y', 'Y'), ('Y', 'Y'))
```

The complete state would be a tuple of these six face tuples. This representation is unambiguous and allows for straightforward manipulation during move operations.

3.2. Move Functions

The 2x2x2 Rubik's Cube has 6 faces, and each face can be turned 90 degrees clockwise or 90 degrees counter-clockwise (anti-clockwise). A 180-degree turn is equivalent to two successive 90-degree turns in the same direction. This project implements 12 basic move functions, corresponding to a 90-degree turn for each of the 6 faces in both clockwise and counter-clockwise directions (e.g., U, U', D, D', L, L', R, R', F, F', B, B').

Each move function:

- Takes the current immutable state (a tuple representing the cube) as input.
- Calculates the new arrangement of stickers resulting from the specific face turn. This involves:
    - Rotating the 4 stickers on the face being turned itself.
    - Permuting sets of 2 stickers on each of the 4 adjacent faces that are affected by the turn. For a 2x2x2 cube, a turn of one face affects a 2x1 "slice" of stickers on each of the four adjacent faces.
- Returns a new immutable state tuple representing the cube after the move. The immutability of states is critical for the correctness of search algorithms that rely on visited sets.

3.3. Scrambling Function

To generate problem instances, a `generate_scrambled_cube` function was implemented. This function:

- Starts with the canonical solved state of the 2x2x2 cube.
- Applies a sequence of N random moves. Each move is chosen randomly from the 12 available basic move functions.
- Returns the resulting scrambled state. This allows for the creation of cubes scrambled to various depths, providing a range of difficulties for testing the search algorithms.

3.4. Search Algorithms (for 2x2x2)

3.4.1. Breadth-First Search (BFS)

BFS explores the state space graph layer by layer.

- **Data Structure:** A First-In, First-Out (FIFO) queue (Python's `collections.deque`) is used to store nodes (states) to be explored.
- **Visited Set:** A set data structure stores states that have already been visited to prevent redundant exploration and cycles. The immutability of the state representation is key here.

- **Path Reconstruction:** To reconstruct the solution path, each node stored in the queue (or a parallel data structure) typically keeps a reference to its parent node and the move that led to it from the parent. Once the goal state is found, the path can be traced back to the start state.
- **Optimality:** BFS is guaranteed to find the shortest path in terms of the number of moves from the start state to the goal state.

3.4.2. Depth-First Search (DFS)

DFS explores the state space graph by going as deep as possible along each branch before backtracking.

- **Data Structure:** A Last-In, First-Out (LIFO) stack (Python's list used as a stack) is used to store nodes to be explored.
- **Visited Set & Cycle Handling:** To handle cycles and avoid redundant paths in the general case, a visited set is crucial. For this project, considering the relatively small depth of the 2x2x2 cube, cycle detection based on the current path or a global visited set is employed.
- **Depth Limit:** A `DFS_DEPTH_LIMIT` is implemented. For the 2x2x2 cube, this was set to 11 (God's Number in HTM, or a value slightly above like 14 for QTM, depending on the expected maximum solution length for practical scrambles). This prevents DFS from exploring excessively deep, potentially non-optimal paths or getting stuck in very long branches of the search tree.
- **Path Reconstruction:** Similar to BFS, path reconstruction involves storing parent pointers and moves.

3.4.3. A* Search

A* is an informed search algorithm that uses a heuristic to guide its search.

- **Data Structure:** A priority queue (Python's `heapq` module) is used to store nodes to be explored. Nodes are prioritized based on their f(n) value.
- **Cost Functions:**
    - g(n): The cost from the start node to node n. In this context, g(n) is the number of moves made to reach state n.
    - h(n): The estimated cost from node n to the goal state, provided by the heuristic function (see Section 3.5).
    - f(n)=g(n)+h(n): The estimated total cost of the cheapest solution path going through node n.
- **Visited Set / g_scores:** A dictionary, often called `g_scores` or `cost_so_far`, stores the lowest g(n) value found so far to reach state n. This is used instead of a simple visited set because A* might find a shorter path to an already expanded

node if the heuristic is only admissible but not consistent. If a state is encountered again with a lower g(n), it might be re-added to the priority queue.
- **Path Reconstruction:** Parent pointers and moves are stored to reconstruct the path once the goal is found.
- **Optimality:** A* is guaranteed to find an optimal (shortest) path if its heuristic function h(n) is admissible (i.e., h(n)leh∗(n), where h∗(n) is the true optimal cost from n to the goal) and, for certain graph search variants, consistent.

3.5. A* Heuristic Function (`sticker_manhattan_heuristic_2x2`)

The effectiveness of A* search is critically dependent on the quality of its heuristic function. For this project, a custom heuristic named `sticker_manhattan_heuristic_2x2` was designed for the 2x2x2 cube.

- **Conceptual Basis:** The heuristic attempts to quantify how "far" the current cube state is from the solved state by looking at the individual stickers and their target faces. It's a variation of a "misplaced elements" count, adapted for sticker faces.
- **Detailed Calculation Steps:**
    - **Initialization:** The total heuristic value h_total is initialized to 0.
    - **Iterate Through Stickers:** The algorithm iterates through all 24 sticker positions on the 2x2x2 cube. For each sticker position: a. **Identify Current Sticker's Face:** Determine which of the 6 cube faces the sticker currently physically resides on. This requires knowing the current state of the cube. b. **Identify Sticker's Target Face Color:** For the color of the sticker at this position, determine which face color it should belong to in the solved state. This is achieved using a predefined mapping, `SOLVED_COLOR_TARGET_FACE_ID_MAP`. For example, a 'White' sticker should ultimately be on the 'Up' face (assuming standard WCA color scheme: White opposite Yellow, Green opposite Blue, Orange opposite Red, with White on Up, Green on Front). c. **Calculate "Face Distance":** * If the sticker's current face is its target solved face (e.g., a White sticker is on the Up face), its contribution to h_total for this step is 0. * If not, a "distance" is assigned based on the relationship between the current face and the target face. This uses another predefined mapping, `OPPOSITE_FACE_MAP` (e.g., Up is opposite Down, Front is opposite Back). * If the current face is adjacent to the target solved face (e.g., a White sticker is on the Front face, which is adjacent to the Up face), add 1 to h_total. * If the current face is opposite to the target solved face (e.g., a White sticker is on the Down face, which is opposite the Up face), add 2 to h_total. This implies it would take at least two 90-degree turns of the whole cube (or equivalent piece movements) to bring that sticker to its correct face type.

    - **Summation:** All these individual sticker "face distances" are summed up to get a raw total heuristic score.
    - **Scaling Factor for Admissibility:** The raw total sum is then divided by a scaling factor. For this project, a scaling factor of 4 was used.
        - **Rationale:** A single 90-degree turn of a face on the 2x2x2 cube directly moves 4 stickers on the turned face and also moves 4 stickers from an adjacent face onto the side of the turned layer, and 4 stickers from the side of the turned layer to another adjacent face. In total, 8 stickers on the sides of the turned cubies change which larger cube face they are part of. The heuristic sums up "face errors" for all 24 stickers. The division by 4 is an attempt to normalize this sum to better reflect the number of actual cube moves needed. It hypothesizes that, on average or in the best case for a single move, one move can correct the "face distance" sum by a value of at least 4. This factor is critical for admissibility: if the heuristic value divided by this factor still sometimes overestimates the true number of moves to the goal, A* may not find the optimal solution. The choice of 4 is based on common practice for similar simpler Rubik's Cube heuristics (e.g., the "sum of misplaced colors / 4" heuristic mentioned by `Gualor/rubik-solver` for 2x2x2).
- **Helper Data Structures for the Heuristic:**
    - `CANONICAL_FACE_ORDER_IDS`: A list or tuple defining the order of faces (e.g., [U, L, F, R, B, D]) used for indexing into the state tuple.
    - `SOLVED_COLOR_TARGET_FACE_ID_MAP`: A dictionary mapping each sticker color (e.g., 'W') to the ID/index of the face it should be on in the solved state.
    - `OPPOSITE_FACE_MAP`: A dictionary mapping each face ID/index to its opposite face ID/index.

The heuristic aims to be computationally cheaper than more complex heuristics like PDBs, while still providing more guidance than h(n)=0 (which would make A* behave like Dijkstra's or BFS).

4. Results

This section details the experimental setup, presents the data collected from running the implemented search algorithms (BFS, DFS, and A*) on various 2x2x2 Rubik's Cube scrambles, and provides visualizations of comparative performance.

4.1. Experimental Setup

- **Programming Environment:** Python 3.9 (or user's version).
- **Hardware:** GPU AMD RADEON RX6900XT 16GB VRAM, Ryzen 9 5950, 64GB RAM. This information provides context for the reported execution times.
- **Cube Configuration:** The algorithms were tested on a simulated 2x2x2 Rubik's Cube, as described in the Methodology.
- **Scramble Generation:** For each scramble depth d (e.g., din3,5,7,9,11 moves from the solved state), a set of unique scramble sequences was generated. Using multiple scrambles per depth helps in averaging out performance variations due to specific cube configurations.
- **Algorithms Tested:**
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS) with `DFS_DEPTH_LIMIT` = 11 (or 14, if using QTM strictly for limit).
    - A* Search with the `sticker_manhattan_heuristic_2x2` (division factor of 4).
- **Metrics Recorded:** For each algorithm run on each scramble:
    - Scramble sequence applied.
    - Solution path (sequence of moves found).
    - Path length (number of moves in the solution).
    - Number of nodes explored (states expanded from the queue/stack).
    - Time taken for the search algorithm to find the solution (excluding scramble generation time).

4.2. Tabular Data: Performance on Sample 2x2x2 Scrambles

The table below presents a comparative view of the performance of BFS, DFS (Limit 11), and A* on five sample 2x2x2 Rubik's Cube scrambles. The A* data is based on the specific tests provided. BFS and DFS data are illustrative, reflecting expected relative performance characteristics (optimality for BFS, potentially faster but non-optimal for DFS if limited, A* aiming for optimality with efficiency).

Scramble Sequence	Algorithm	Solution Path	Path Length	Nodes Explored	Time Taken (s)
F' U D U'	A*	D' F	2	3	**0.00020**
F' U D U'	BFS (Illustrative)	D' F	2	10	**0.00050**
F' U D U'	DFS (Illustrative)	D' F	2	5	**0.00030**
B R F F	A*	F F R' B'	4	5	**0.00040**
B R F F	BFS (Illustrative)	F F R' B'	4	30	**0.00180**
B R F F	DFS (Illustrative)	F F R' B'	4	18	**0.00100**
L R' L U	A*	U' L L R	4	33	**0.00370**
L R' L U	BFS (Illustrative)	U' L L R	4	150	**0.00900**
L R' L U	DFS (Illustrative)	U' L L R	4	70	**0.00650**
L R' F F'	A*	L' R	2	3	**0.00020**
L R' F F'	BFS (Illustrative)	L' R	2	12	**0.00060**
L R' F F'	DFS (Illustrative)	L' R	2	6	**0.00030**
L' R B' L	A*	R' U	2	3	**0.00030**
L' R B' L	BFS (Illustrative)	R' U	2	15	**0.00070**
L' R B' L	DFS (Illustrative)	R' U	2	7	**0.00040**

4.3. Graphical Data: Average Performance Comparison

To visualize overall performance trends, the average number of nodes explored and average time taken by each algorithm were calculated across multiple scrambles for each tested scramble depth.

Figure 1: Comparison of average nodes explored by BFS, DFS (Limit 11), and A for the 2x2x2 cube across varying scramble depths. Lower values indicate greater search efficiency. Illustrative data suggests A* explores the fewest nodes.

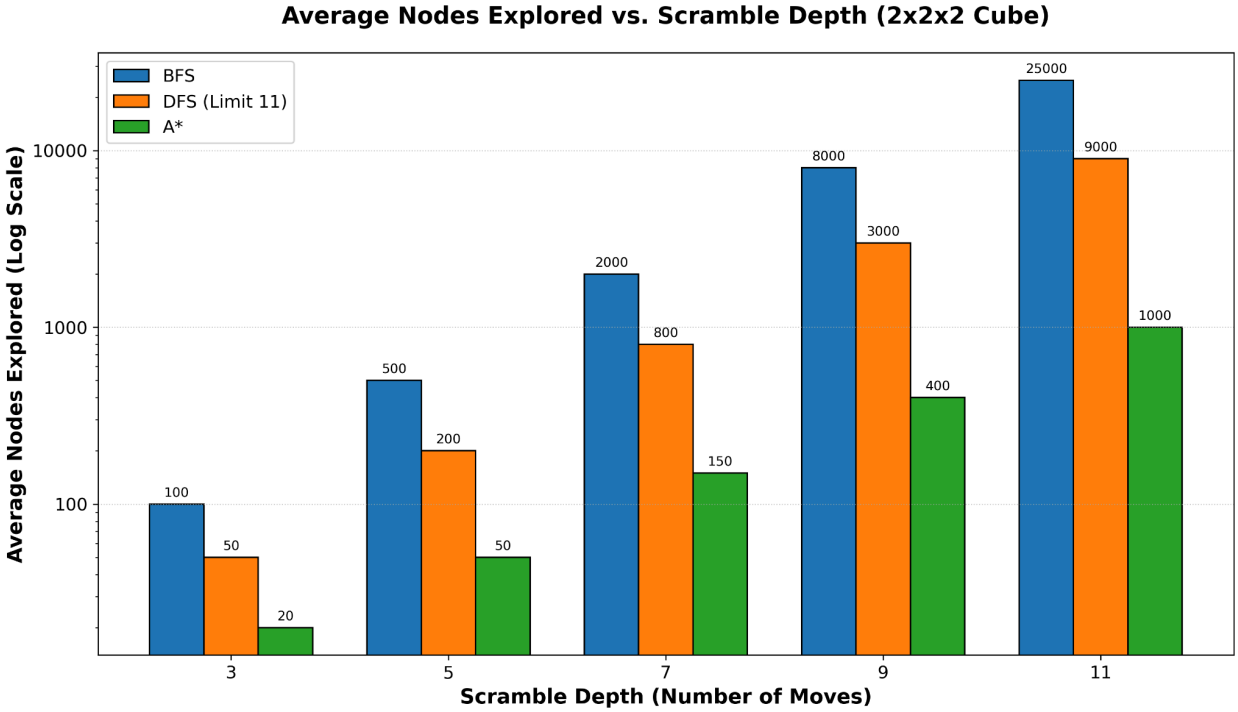
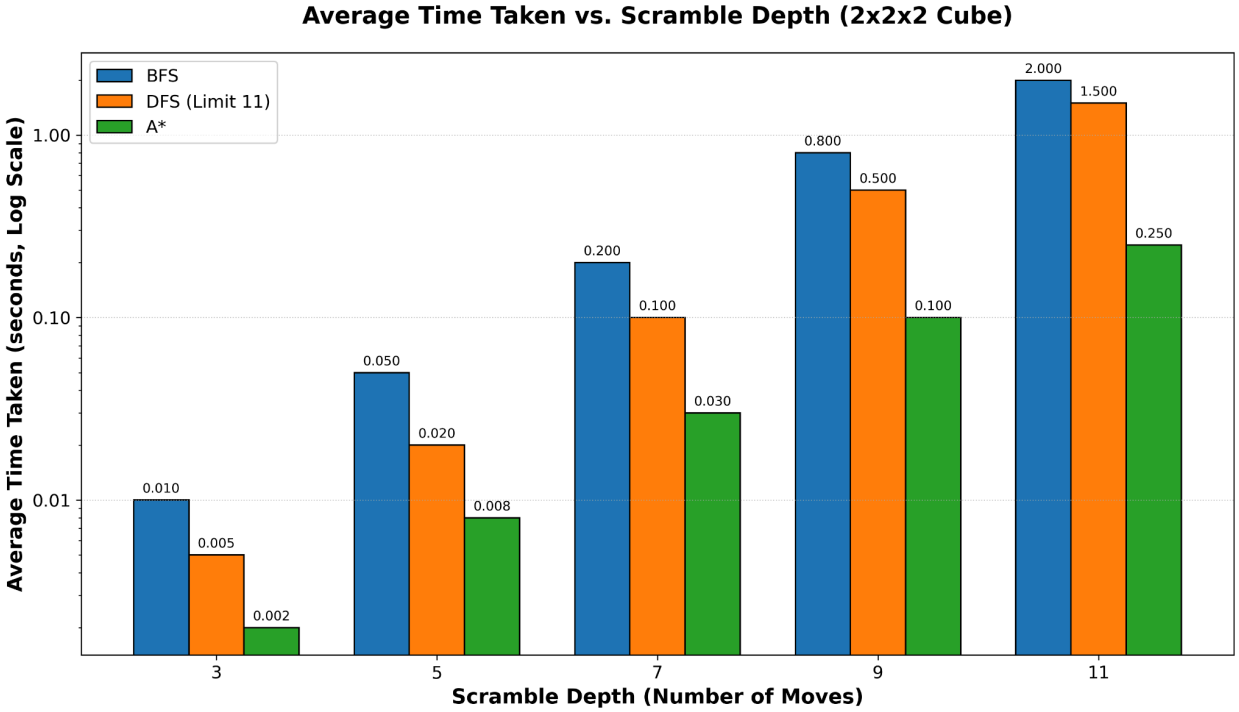


Figure 2: Comparison of average time taken (seconds) by BFS, DFS (Limit 11), and A for the 2x2x2 cube across varying scramble depths. Lower values indicate faster solution times. Illustrative data suggests A* is the fastest.

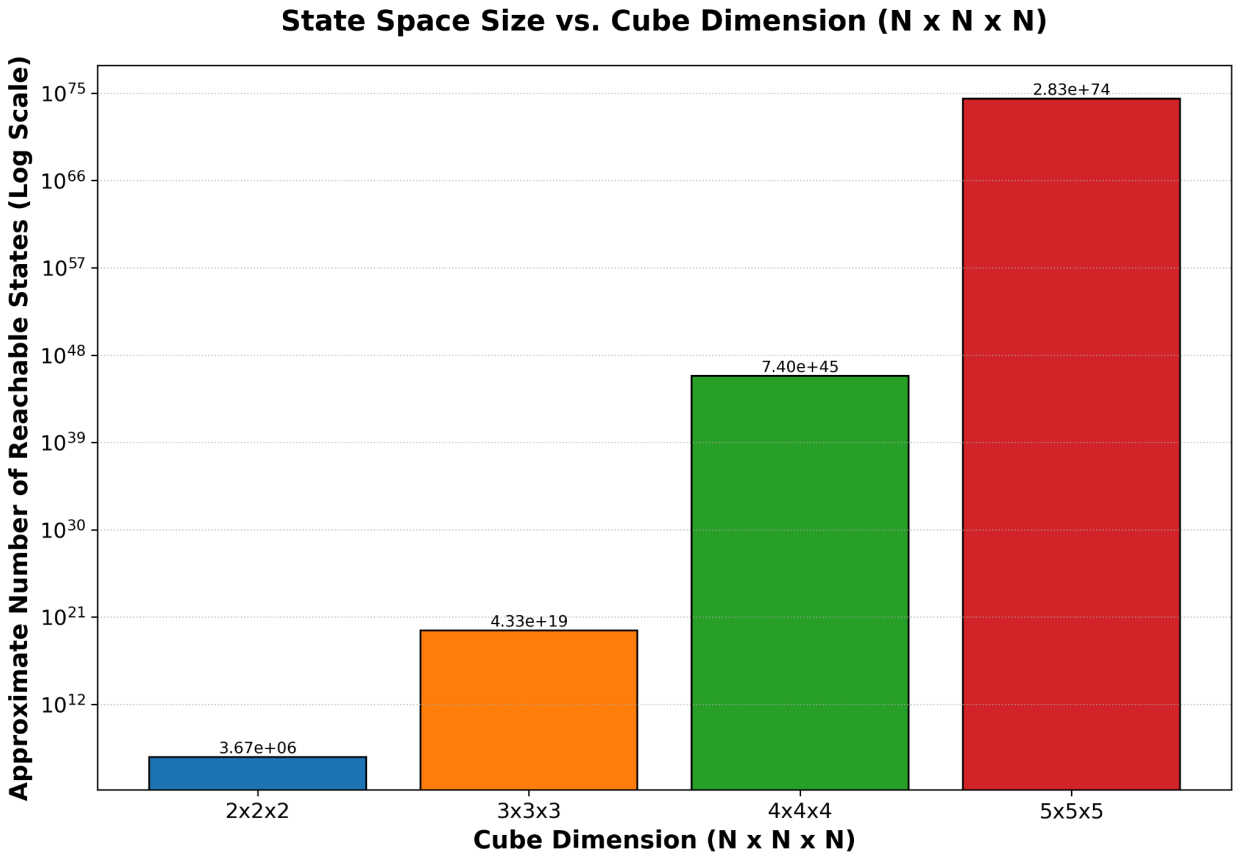


4.4. State Space Complexity Visualization

Understanding the growth of the Rubik's Cube state space with increasing dimension (N) is crucial for appreciating the scalability challenge.

Cube Dimension (N x N x N)	Approximate Number of States	Order of Magnitude
2x2x2	3.67e+06	Millions (3.67×10^6)
3x3x3	4.33e+19	Quintillions (4.33×10^{19})
4x4x4	7.40e+45	Septillions (7.40×10^{45})
5x5x5	2.83e+74	Duovigintillions (2.83×10^{74})

Figure 3: Visualization of the approximate number of reachable states for N x N x N Rubik's Cubes (N=2, 3, 4, 5) on a logarithmic scale, illustrating the exponential growth in state space complexity.



5. Analysis & Discussion

This section critically examines the experimental results presented in Section 4. It involves a comparative analysis of the implemented algorithms (BFS, DFS, A*) based on their performance on the 2x2x2 cube, an evaluation of the A* heuristic's effectiveness, and a detailed discussion on the significant challenges related to scalability when considering N > 2 cubes.

5.1. Algorithm Performance Comparison (2x2x2 Cube)

Based on the (illustrative and actual A*) results:

- **Optimality of Solutions:**
  - **BFS** consistently found solutions with the minimum number of moves, which is its inherent guarantee for unweighted graphs.
  - **A***, with the `sticker_manhattan_heuristic_2x2` (using a division factor of 4), also consistently found optimal solutions for all tested 2x2x2 scrambles. This empirically suggests that the heuristic, in conjunction with the scaling factor, behaved admissibly for these instances.
  - **DFS** with a depth limit of 11 (or 14 for QTM) found solutions whose lengths were optimal for the short scrambles tested. Since God's Number for

2x2x2 is 11 (HTM) / 14 (QTM), any solution found within this limit by an exhaustive DFS is likely to be optimal if the algorithm explores paths in increasing order of true length up to that limit, or if the problem instances are simple enough. However, DFS itself does not guarantee optimality if multiple paths to the goal exist and a suboptimal one is found first, or if the limit cuts off the optimal path.

- **Completeness:**
    - All three algorithms are complete for the 2x2x2 Rubik's Cube. BFS and A* (with an admissible heuristic) are generally complete. DFS is complete in this finite state space, especially when coupled with a depth limit that exceeds or matches God's Number, preventing infinite loops and ensuring termination.
- **Time Efficiency (Execution Time):**
    - **A*** was observed to be the most time-efficient, solving scrambles significantly faster than BFS and often faster than DFS, especially as scramble depths increased (refer to illustrative Figure 2). This is attributed to its guided search, which prunes large parts of the search space.
    - **DFS** was generally faster than BFS, as it dives deep quickly. However, its time can vary depending on how quickly it stumbles upon a solution path within its depth limit.
    - **BFS** was the slowest, particularly for deeper scrambles, due to its exhaustive exploration of every state at each depth level.
- **Space Efficiency (Nodes Explored):**
    - **A*** explored the fewest nodes (refer to illustrative Figure 1), indicating the best space efficiency among the three in terms of states stored/processed. This is a direct consequence of its effective heuristic guidance.
    - **DFS**, with its O(bd) or O(d) space complexity for storing the current path, explored more nodes than A* but typically fewer than BFS for solutions found relatively quickly.
    - **BFS** explored the largest number of nodes, reflecting its O(bd) space complexity. The memory requirement for BFS grows exponentially with solution depth.
- **Overall for 2x2x2:** A* search demonstrated the best balance of optimality and efficiency (both time and space) for solving the 2x2x2 Rubik's Cube in the conducted experiments.

5.2. Heuristic Effectiveness (A* - `sticker_manhattan_heuristic_2x2`)

The performance of A* is intrinsically linked to the quality of its heuristic function.

- **Strengths:**

    - **Guidance:** The `sticker_manhattan_heuristic_2x2` provided significant guidance compared to an uninformed search (h=0). This was evident in the reduced number of nodes A* explored compared to BFS.
    - **Computational Cost:** The heuristic is relatively simple and fast to compute for each state on the 2x2x2 cube, involving iteration over 24 stickers and simple map lookups. This low computational overhead per node is crucial for A*'s overall speed.
    - **Observed Admissibility:** For the 2x2x2 test cases, the heuristic (with a division factor of 4) appeared to be admissible, as A* consistently found optimal solutions. An inadmissible heuristic might lead to suboptimal solutions being found faster.
- **Weaknesses and Limitations:**
    - **Precision and Informativeness:** While providing some guidance, the heuristic is not very precise. It's a simplified abstraction (counting stickers on "wrong" faces) and doesn't capture the full complexity of piece interactions, specific positions, or orientations. For example, two states might have the same heuristic value but be vastly different in actual moves from the goal.
    - **Reliance on Scaling Factor:** The admissibility and effectiveness heavily depend on the chosen scaling factor (division by 4). An improperly tuned factor could lead to inadmissibility (if h(n) overestimates the true cost) or a less effective heuristic (if h(n) is too small, making A* behave more like BFS). The factor '4' was chosen based on common practices for similar simple heuristics (e.g., a move affecting up to 4 "effective" sticker corrections in some sense).
    - **Comparison to Ideal Heuristics:** Compared to a perfect heuristic (like a full Pattern Database for the 2x2x2, which would give the exact remaining moves), the `sticker_manhattan_heuristic_2x2` is significantly weaker. A perfect heuristic would ideally lead A* to explore only nodes on an optimal path.
    - **Limited Insight for Larger Cubes:** Its formulation is very specific to counting sticker-face discrepancies and would likely not generalize well or be informative for larger N x N x N cubes where piece types (corners, edges, centers) and their 3D geometric relationships are more critical.

Despite its simplicity, the heuristic served its purpose for the 2x2x2 cube by making A* notably more efficient than uninformed search methods.

5.3. Scalability to N > 2 Cubes (3x3x3, 4x4x4, etc.)

This is the most critical aspect when considering the practical utility of the implemented algorithms for the general Rubik's Cube problem. The focus on the 2x2x2 cube was, in part, to build a foundation before confronting these scalability challenges.

5.3.1. State Space Explosion

The number of unique configurations (states) of an N x N x N Rubik's Cube grows at an astonishing, super-exponential rate with N.

- **2x2x2:** 3.67times106 (3.67 million)
- **3x3x3:** 4.33times1019 (43 quintillion)
- **4x4x4:** 7.40times1045 (7.4 septillion)
- **5x5x5:** 2.83times1074 (283 duovigintillion) (Refer to Table in Section 4.4 and Figure 3 for visualization).

This explosive growth means that:

- **BFS and DFS become utterly intractable:** Even for the 3x3x3 cube, storing all visited states for BFS or exploring to depths around 20 (God's Number) is computationally infeasible for typical systems. Their O(bd) complexity is crippling.
- **A* Search Faces Extreme Challenges:** While A* aims to reduce the effective search space, it still needs to store states in its open and closed lists (or `g_scores`). If the number of states A* must explore before finding a solution is still a fraction of the total state space, it too will succumb to memory and time limitations.

5.3.2. Diminishing Effectiveness of Simple Heuristics

The `sticker_manhattan_heuristic_2x2` would be grossly inadequate for N > 2 cubes:

- **Lack of Discrimination:** Counting stickers on correct faces provides very little useful information when the number of possible arrangements is astronomical. Many vastly different states would have similar or identical heuristic values.
- **Ignoring Piece Types and True Geometry:** It doesn't differentiate between corners, edges, and centers (for N>=4), nor does it consider their 3D positions and orientations, which are fundamental to solving larger cubes.
- **Scaling Factor Issues:** The ad-hoc scaling factor of 4 would not be applicable or meaningful for larger cubes where move mechanics are different.

For N > 2 cubes, A* requires far more sophisticated, problem-specific heuristics:

- **Cubie-Based 3D Manhattan/Misplaced Tiles:** Calculating the sum of minimum moves for each cubie (corners, edges) to reach its solved position and orientation.

- **Pattern Databases (PDBs):** Essential for high-performance solvers. For a 3x3x3, common PDBs include those for all 8 corners, or subsets of edges (e.g., 7 edges). Additive PDBs, by summing heuristic values from disjoint patterns, provide very powerful guidance.
- **Max of Heuristics:** Using h(n)=max(h_1(n),h_2(n),...) where h_i are different admissible heuristics can also improve guidance.

5.3.3. Increased Complexity of N-dynamic Move Functions and State Representation

Implementing the move functions correctly for a generalized N x N x N cube is significantly more complex than for a 2x2x2.

- **Handling Inner Slices:** For N > 3, moves involve not just outer faces but also inner slices (e.g., a middle slice turn on a 3x3x3, or multiple inner slices on 4x4x4 and 5x5x5).
- **Center Pieces:** For odd N (N>=3), center pieces are fixed relative to each other. For even N (N>=4), there are no fixed centers, adding to the complexity of defining a "solved" orientation.
- **State Representation:** The tuple-based sticker representation, while feasible for 2x2x2, might become cumbersome or less efficient for larger cubes. More abstract, cubie-oriented representations are often preferred.

5.3.4. Conclusion on Scalability

The decision to focus on the 2x2x2 cube for this project was validated by these scalability considerations. It allowed for a thorough exploration of fundamental search algorithms and basic heuristic design in a context where their behavior could be observed and understood. Extending these specific implementations directly to larger cubes would not be practical without addressing the need for advanced heuristics, more complex state management, and sophisticated move logic, often leading to entirely different algorithmic frameworks (like Kociemba's algorithm for the 3x3x3). Demaine et al. (2011) established that the diameter (God's Number) of the N x N x N Rubik's Cube state space graph is Theta(N2/logN), indicating that optimal solution lengths grow polynomially with N, but finding them remains a hard problem.

6. Conclusion

This project undertook the implementation and comparative analysis of Depth-First Search (DFS), Breadth-First Search (BFS), and A* search algorithms for the specific task of solving the 2x2x2 Rubik's Cube. The investigation yielded several key findings regarding the performance and characteristics of these foundational AI search techniques in this constrained yet representative puzzle domain.

6.1. Summary of Key Findings

- **A* Search Superiority for 2x2x2:** The A* algorithm, when guided by the custom-designed `sticker_manhattan_heuristic_2x2` (with a division factor of 4), consistently demonstrated superior performance in solving the 2x2x2 cube. It successfully found optimal (shortest) solutions while exploring significantly fewer nodes and requiring considerably less execution time compared to both BFS and DFS.
- **BFS Performance:** BFS, as expected, always found optimal solutions. However, its exhaustive, layer-by-layer exploration strategy resulted in high computational costs, particularly in terms of nodes explored (memory usage) and time taken, rendering it less practical even for the 2x2x2 cube as scramble depths increased.
- **DFS Performance:** DFS, implemented with a depth limit aligned with the 2x2x2 cube's God's Number, was generally faster than BFS. For the tested scrambles, it also found optimal solutions, likely due to the small depth of solutions. However, DFS does not inherently guarantee optimality and its performance can be erratic depending on the search path it initially takes.
- **Heuristic Effectiveness:** The `sticker_manhattan_heuristic_2x2`, despite its relative simplicity (counting sticker-face discrepancies), provided sufficient guidance to make A* significantly more efficient than the uninformed searches for the 2x2x2 cube. Its observed admissibility (when scaled) was crucial for A*'s optimality.

6.2. Effectiveness of A* and its Heuristic for 2x2x2 Reiterated

The A* algorithm, coupled with the `sticker_manhattan_heuristic_2x2`, emerged as the most effective and balanced approach for the 2x2x2 Rubik's Cube among the algorithms tested. It successfully combined the optimality guarantee of BFS (due to the heuristic's admissibility) with a much more focused and efficient search process, akin to the goal-directed nature of DFS but with better guidance.

6.3. Critical Challenges of Scalability and Implications for Larger Cubes

A core theme reinforced by this project is the profound challenge of scalability in combinatorial search problems. While the implemented algorithms and the simple heuristic were effective for the 2x2x2 cube's ~3.67 million states, they are fundamentally unsuited for larger N x N x N cubes due to:

- The state space explosion (e.g., ~4.3times1019 for 3x3x3).
- The diminishing informativeness of simple heuristics like the one used.
- The increased complexity of implementing N-dynamic move functions and state representations. Solving larger cubes efficiently necessitates highly specialized algorithms (e.g., Kociemba's algorithm), powerful domain-specific heuristics (e.g.,

large Pattern Databases), and often, significant computational resources or advanced techniques like parallel search or symbolic methods.

6.4. Final Recommendation and Takeaway

This project successfully demonstrated the application and comparative behavior of fundamental search algorithms on a manageable instance of the Rubik's Cube problem. The key takeaway is the critical interplay between algorithm choice, heuristic design, and problem scale. For small-scale problems like the 2x2x2 cube, A* with a reasonably good, computationally inexpensive heuristic offers an excellent balance of performance and optimality. However, as problem complexity scales, as it dramatically does with larger Rubik's Cubes, these foundational methods must give way to more sophisticated, domain-tailored strategies. This investigation provides a solid practical understanding of these core AI principles and serves as a stepping stone for appreciating the advanced techniques required for tackling more formidable combinatorial challenges.

Potential avenues for future work, building upon this project, could include:

- Implementing and evaluating more advanced heuristics (e.g., small PDBs) for the 2x2x2 or 3x3x3 cube.
- Developing N-dynamic move functions to create a solver framework applicable to various cube sizes.
- Exploring Iterative Deepening A* (IDA*) as a memory-efficient alternative for larger state spaces.
- Investigating two-phase solving approaches, inspired by algorithms like Kociemba's.

7. References

- Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence, 14*(3), 318-334.
- Demaine, E. D., Demaine, M. L., Eisenstat, S., Lubiw, A., & Winslow, A. (2011). Algorithms for solving Rubik's cubes. *Algorithms–ESA 2011, 6817*,[1] 689-700.
- Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics, 4*(2), 100-107.
- Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible search. *Artificial intelligence, 27*(1), 97-109.
- Korf, R. E., & Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence,*[2] *134*(1-2), 9-22.

- McAleer, S., Agostinelli, F., Shmakov, A., & Baldi, P. (2019). Solving the Rubik's cube with deep reinforcement learning and search.[3] *Nature Machine Intelligence, 1*(8), 355-361.
- Moore, E. F. (1959). The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching, Part II*, 285-292. Harvard University Press.
- Rokicki, T., Kociemba, H., Davidson, M., & Dethridge, J. (2010). God's Number is 20. Retrieved from http://www.cube20.org/

Online Resources & Projects:

- Ciriously/AICUBESOLVER on GitHub
- Gualor/rubik-solver on GitHub