An Analysis of Rubik's Cube Solving Algorithms

Anca Ionela Nicu
University of Craiova
Craiova, Romania
ancanicu2001@gmail.com

Paul Stefan Popescu
University of Craiova
Craiova, Romania
stefan.popescu@edu.ucv.ro

ABSTRACT

Nowadays, there are numerous methods to solve the Rubik's Cube of various dimensions, most of which aim to find the optimal solution. Our application is an addition to them, focusing on discovering a solution to the pocket Rubik's Cube by applying six different algorithms and comparing their efficiency. Each of the six methods has been run four times on a set of 45 test data (five cube configurations for each optimal solution length from one to nine, where the optimal solution represents the shortest sequence of moves that can solve a cube), and for each test data, we computed the success rate in discovering a solution, the average memory used, the average execution time, and the average solution length (the average number of moves a solution has). Based on these results, we grouped the findings by the optimal solution length and compared the algorithms on the four criteria mentioned earlier. Comparing the algorithms' results, we find that the Layer-by-layer method is the best in terms of memory, execution time and success rate, but the worst in terms of solution length, A* and bidirectional BFS (Breadth-first search) are good overall for each criterion considered, DFS (Depth-first search) has the lowest success rate overall, and MCTS (Monte Carlo Tree Search) and Q-learning do not perform well, especially on complex cube configurations (cubes that can be solved optimally on six to nine moves). This research contributes to the field of Rubik's cube solvers by developing six different ways from scratch to solve the 2x2 cube and by demonstrating their efficiency on different cube configurations, in comparison to each other.

Author Keywords

Rubik's cube, algorithms comparison, bidirectional BFS, DFS, MCTS, Q-learning, A*, Layer-by-layer

General Terms

Human Factors; Design; Measurement.

DOI: 10.37789/icusi.2025.23

INTRODUCTION

Created fifty years ago, the Rubik's Cube remains a popular toy. Since its creation, people have sought innovative ways to solve the cube physically. Those methods evolved and were tailored for different purposes: some of them imply memorizing fewer algorithms and are suitable for beginners (such as Layer-by-layer), and others are made for speed cubing and need memorizing a lot of algorithms (methods like CFOP, Petrus, or Roux). Alongside technological development, computer scientists attempted to find ways to solve the cube in as few rotations as possible.

Using computers provided by Google, a team of researchers established in 2010 that any 3x3 Rubik's Cube can be solved in a maximum of 20 moves.[12] This number is called *God's number*, and there are studies [5] demonstrating that it grows as $\Theta(n2/\log n)$ for an n×n cube. For a 2x2 cube (the object of our study), *God's number* is 11 when using the half-turn metric.

Determining an optimal solution (or even a solution for the bigger cube sizes) is not an easy task, especially considering the huge state space (that grows considerably alongside with the size of the cube). So, building algorithms for the Rubik's cube implies finding a balance both between memory and time and between success rate and solution length. A method that solves the memory problem is the one developed by Kunkle and Cooperman's ("Disk is the New RAM" [9]), in which they showed that, by using commodity disks and their Roomy library, a 3x3x3 cube can be solved in ≤ 26 moves, which is pretty close to the optimal solution length. However, their approach sacrifices the time component, needing a long time for parallel preprocessing.

Used increasingly often nowadays in almost every aspect of our lives, Artificial Intelligence could definitely be a valid approach for the Rubik's Cube problem. Some models of Reinforcement Learning were already developed by researchers, among which we mention two: one that doesn't use handcrafted heuristics, developed by McAleer et al. [3], and one that extends it with bidirectional value improvement and A* search, developed by Agostinelli et al. [4].

Nowadays, there are plenty of sites and applications for solving the Rubik's cube, each of them with its own characteristics and goals: some of them requires scanning a physical cube and determining a solution for that, others are pure simulators that displays a 3D or 2D representations and allows the user to shuffle the cube randomly or manually, and then returns the solution. Most of the current applications aim to solve the smaller Rubik's cubes (3x3 or 4x4) and focus on applying a single algorithm to get the solution. Moreover, most of them are not intuitive enough or appealing enough.

We ask if known algorithms – such as graph traversal methods (BFS or DFS), machine learning methods (MCTS , Q-learning) or informed search methods (A*) – can be successfully adapted to solving the pocket Rubik's cube, and how a technique used in physically solving the cube (Layer-by-layer) can be implemented in a program. Our hypotheses focus on each of those algorithms as follows:

- Bidirectional BFS will always find the optimal solution, but it will be memory-intensive for the complex configuration.
- DFS will not perform well;
- The efficiency of A* depends on the heuristic and by choosing a good heuristic, A* will provide solutions close to the optimal in terms of length;
- MCTS and Q-learning will need a big amount of training and a good reward function in order to be able to learn how to solve the cube;
- Layer-by-layer will find a long solution compared to the others, but it will be better than them in terms of memory and execution time.

Moreover, we ask if we can create an application that strikes a balance between user-friendly appearance and a robust set of functional algorithms.

In contrast to other applications or sites for solving the Rubik's Cube, our approach focuses on a single Rubik's Cube (the pocket one) and develops six different ways to solve it. The goal of our project is to create an application that is both user-friendly (intuitive and visually appealing) and allows us to compare six algorithms.

In order to make it intuitive, we created a simple scene that contains both the 3D (the actual cube that can be shuffled and solved) and 2D (the cube map that displays the actual cube changes in real time) cube representation, and separate buttons for each operation that can be applied to the cube (one for each solving algorithm and one for randomly shuffling the cube).

Most of the other applications and sites for this field focus on bigger cube sizes (3x3 or higher), but our approach aims to solve the smallest one, which, by its smaller state space, allows us to develop and compare a decent amount of methods to solve it. The algorithms chosen for the comparison are well known and the key for adapting them to the problem was finding a way to simulate the moves and to represent the cube state. More than that, A* and machine learning methods required finding a good heuristic and reward function, respectively.

RELATED WORK

Automatically solving the Rubik's Cube is an enjoyable task that has been addressed in numerous papers, and despite being an old problem, it remains relevant today. Rubik's Cube research encompasses visual analytics, heuristic and learning-based solvers, computational complexity theory, and mechanical and robotic manipulation.

Regarding the visualization of Rubik's cube solution algorithms, Steinparz et al. [1] introduced an interactive projection of high-dimensional cube states that lets analysts see how different solution algorithms traverse the search space; their t-SNE views expose clusters of equivalent sub-problems and reveal why some human methods stall early in the state graph. This work inherits their agenda of human-interpretable debugging but focuses on learning-based solvers whose internal heuristics are opaque without additional instrumentation.

Another approach is using swarm intelligence [11] for solving the Rubik's cube [2] as presented by Jeevan & Nair benchmark four nature-inspired optimizers—PSO, ACO, Krill-Herd and a Greedy Tree Search—on the 3×3×3 cube, reporting success rates below 30 %. Median move counts above 100, yet confirming that population-based search can escape local optima unattainable to deterministic heuristics. Their findings motivate our hybrid approach: they combine a fast RL policy (Section 4) with a lightweight swarm layer that diversifies late-stage searches.

The use of reinforcement learning was an approach explored in both our work and other papers. McAleer et al. [3] propose a bootstrapping scheme that trains value and policy networks entirely from self-generated trajectories, achieving a 100% solve rate with a 30-move median, without the use of handcrafted heuristics. Another approach [4] is presented by Agostinelli et al. called DeepCubeA, that extends this with bidirectional value improvement and A* search, pushing the median down to 26 moves and outperforming Kociemba's two-phase solver on random scrambles. Both methods treat the cube as a sparse-reward MDP and highlight two bottlenecks: (i) the expensive roll-outs during training, and (ii) the limited interpretability of the learned heuristics.

Regarding the algorithmic and complexity foundations, there is some early theoretical work by Demaine et al. [5] that proves that God's Number for the n×n×n cube grows as $\Theta(n^2/\log n)$ and gives an asymptotically optimal parallel algorithm that realizes this bound. Subsequently, the same group (and later an independent arXiv version [6]) established NP-completeness for deciding optimal solutions on the general n-cube, via a reduction from Hamiltonian Cycles on grid graphs. These hardness results justify the community's shift from minimal-move guarantees to near-optimal yet scalable heuristics.

The Rubik's Cube solving problem goes even further in mechanical and robotic manipulation, as practical deployment often requires physically executing moves. Higo et al. [7] present a high-speed, vision-guided, multi-fingered hand that can grasp and reorient individual layers at 10 Hz, demonstrating reliable hardware execution of algorithmic plans. Complementing the hardware view, Zeng et al. [8] review broader mechanical applications of the cube—ranging from dexterity benchmarks to encryption primitives—and argue that manipulation speed, accuracy, and robustness are now mature enough to close the sim-to-real loop for autonomous solvers.

Regarding the external memory utilization Kunkle and Cooperman’s “Disk is the New RAM” [9] showed that sheer I/O bandwidth can substitute for DRAM when enumerating gigantic implicit graphs such as the 3×3×3 cube. Their Roomy library strips multi-terabyte pattern-databases across dozens of commodity disks, enabling a breadth-first exploration that proved every state solvable in ≤ 26 moves—tightening the practical upper bound on God’s Number. While this external-memory strategy yields optimal or near-optimal solutions almost instantly at run time, it pays an upfront cost in many hours of parallel preprocessing and is tied to a single cube size.

Another deep learning approach presented by Johnson (2021) [10] proposes a Learned Guidance Function (LGF) that is trained via deep neural networks to predict distance-to-goal from partial cube states. The LGF is updated in stages, each time using solutions found by an evolutionary strategy (ES) to refine the network, thereby boosting ES success from below 20% to above 80% without the use of hand-crafted heuristics. This supervision-based approach is more data-efficient than sparse-reward reinforcement learning, but it still relies on an initial corpus of solved states and struggles with extrapolation to larger cubes. In contrast, we combine self-play RL with a lightweight swarm layer: the RL policy supplies an adaptive value proxy akin to Johnson’s LGF, while the swarm component injects diversity, eliminating the need for pre-solved examples and yielding stronger generalization.

PROPOSED APPROACH

General Overview

Our project materializes in a simple Unity application that focuses on shuffling a pocket Rubik’s cube and solving it by one of the six implemented algorithms. The pipeline of our application, shown in Figure 1, consists of four main steps: user interaction, cube shuffling, cube solving and scene rendering. The flow of our application starts with the user shuffling the cube manually or by pressing the “Shuffle” button. As a result, the rendering module displays the shuffling steps in sequence, culminating in the final scrambled cube. Then, the user can select an algorithm for solving the cube by pressing one of the “Solve” buttons. The selected algorithm attempts to find a solution, and once the method finishes its execution, the rendering module is used again to display the solution steps and a success message (if a solution is found) or an error message (otherwise).

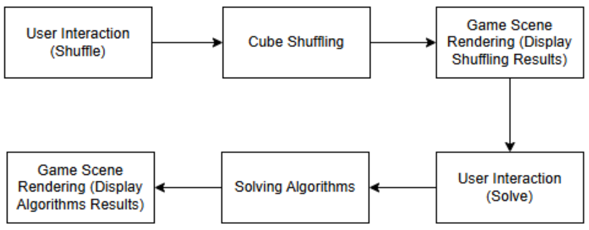
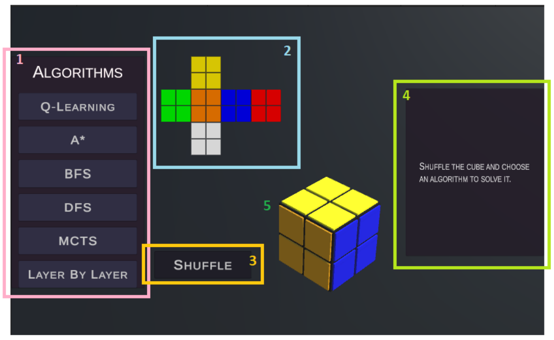

User Interaction (Shuffle)
Cube Shuffling
Game Scene Rendering (Display Shuffling Results)
User Interaction (Solve)
Solving Algorithms
Game Scene Rendering (Display Algorithms Results)

Figure 1 - Application pipeline

The pipeline can be divided into four modules, as follows:

1. **The User Interaction Module** handles the user input, in both shuffling and solving the cube. The shuffling process is triggered by pressing the “Shuffle” button or by manually rotating the faces, whereas the solving process is initiated by pressing one of the solving algorithm buttons.
2. **The Cube Shuffling Module** handles the shuffling cube process that was previously triggered.
3. **The Game Scene Rendering Module** handles the visuals in both shuffling (displaying the shuffling steps and updating the cube map and cube state accordingly) and solving (displaying the selected algorithm results and, for a successful run, updating the cube map and cube state accordingly) processes.
4. **The Solving Algorithms Module** is responsible for applying the selected algorithm to the current cube state. By using raycasting, the current cube state is read and, using it as the initial state, the chosen algorithm steps are applied to it. After the method finishes its execution, the execution time is displayed. If a solution is found, it is then passed to be applied to the cube in the scene.

User Interface



SHUFFLE THE CUBE AND CHOOSE AN ALGORITHM TO SOLVE IT.

Figure 2 - User Interface

The user interface (see Figure 2) is simple and user-friendly. It consists of five areas: Area 1 contains the buttons for solving algorithms, six buttons in total. Area 2 contains the cube map, which is updated in real-time based on the 3D cube representation. Area 3 is the shuffle button, that can be pressed multiple times in a row so that the user gets the cube as scrambled as preferred. Area 4 is the results panel, which displays a different message based on the situation (the one in Figure 2 at the beginning of the game, the name of the running algorithm after a solve button was pressed, or the chosen algorithm results after the method finished its execution). Area 5 is the 3D cube that can be shuffled, solved, and rotated.

Cube Faces Rotations

The cube configuration in our project follows the classical one: red face opposed to the orange one, yellow face opposed to the white one, and green face opposed to the blue one, with red, white and blue faces arranged in this order clockwise (as shown in Figure 2, Area 2). In contrast to the real-world cube, where the faces are considered based on their position in relation to the user, our approach considers the faces based on their original colour. So, for naming the faces, and then the related rotations, we used the following notations: F for the front/orange face, B for back/ red face, U for up/yellow face, D for down/white face, L for left/green face and R for right/ blue face.

Our implementation follows the half-turn metric and uses the Singmaster notation, so we have 18 types of rotations, six 90^0 counterclockwise rotations (F', R', L', U', B', D'), six 90^0 clockwise rotations (F, R, L, U, B, D) and six double rotations (F2, R2, L2, U2, B2, D2). Those rotations are used in both shuffling and solving the cube.

In code, the simulation of rotations was hard coded using functions that get the cube state, apply the rotation logic and return the resulting cube state.

Cube State Representation

In our approach, a cube configuration is represented as a 24-character string, four characters for each face, with yellow, blue, orange, green, red, and white faces in this order, as presented in Figure 3. So, for the cube configuration in Figure 3, the cube state string is "UUUURRRRFFFFDDDDLLLLBBBB".

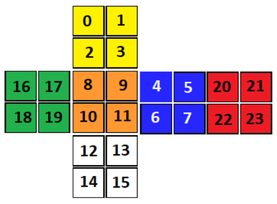


Figure 3 - Cubelets order in cube state representation

The actual cube configuration when pressing a solving button is read from the 3D cube using raycasting and passed to the chosen algorithm in this format.

Solved Cube Conditions

In real life Rubik's cube solving or in other applications or sites, a cube is considered solved when each face contains only one solid colour, and rotating the entire cube doesn't change its configuration. Compared to them, our approach has a more strict set of rules. So, to be considered solved, a cube should have the exact configuration shown in Figure 3 and its state should be "UUUURRRRFFFFDDDDLLLLBBBB". No other configuration will be considered as a solved cube.

ALGORITHMS

Layer-by-layer

The layer-by-layer algorithm implements the steps followed by a beginner in solving the 2x2 Rubik's cube, and consists in four steps:

<u>Step 1</u> solves the yellow face and the row below it. At the end of it, the cube map looks as in Figure 4 (grey cells can be any of the available colours), and the cube state is "UUUURRxxFFxxxxxxLLxxBBxx", where x can be replaced by any of the available faces identifiers.

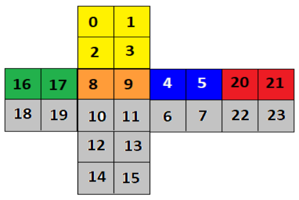


Figure 4 - Cube map after step 1

<u>Step 2</u> (OLL - Orient The Last Layer) solves the white face, but not the adjacent row. At the end of it, the cube map looks as in Figure 5 (grey cells can be any of the available colours), and the cube state is "UUUURRxxFFxxDDDDLLxxBBxx", where x can be replaced by any of the available faces identifiers.

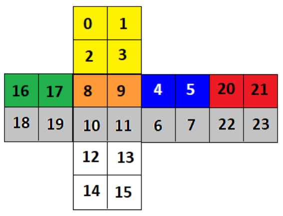


Figure 5 - Cube map after OLL

<u>Step 3</u> (PLL - Permutation of The Last Layer) solves the adjacent layer of the white face (the one coloured with grey in Figure 5). At the end of it, the two median layers (the layer adjacent to white and the one adjacent to yellow in Figure 5) can be misaligned.
<u>Step 4</u> align the two layers by rotating the white one accordingly.

Bidirectional BFS

Even for the pocket Rubik's cube the state space is huge so, in consequence, a classical BFS approach will be space and time consuming. In order to solve this issue, we used a bidirectional BFS approach, that still preserves the BFS property of finding the shortest path (the optimal solution).

The bidirectional BFS approach involves applying classical BFS twice: once starting from the shuffled cube and traversing the tree towards the solved cube (called *forward*) and once starting from the solved cube and traversing the tree towards the shuffled cube (called *backward*). The algorithm stops when a state visited by one traversal was already visited by the other. The solution is built reuniting the paths followed by the two traversals: the sequence of moves in forward traversal stays the same, and it is followed by the sequence of backward traversal, reversed and inverted.

DFS

The DFS approach we used in our implementation is the classical one. Its goal is to find a solution, not the optimal one. Because of the large state space, we made some adjustments: we limited the search depth to 11 (because the God's number is 11 and any of the test data in our experiments can be solved optimally in less than ten moves) and the maximum number of steps to three million.

Moreover, to increase the chances of finding a solution, we randomly shuffled the allowed rotations for each step of the algorithm. In this approach, for a particular step, the allowed moves are all half-turn metric rotations except for rotations of type X, X' or X2, where the move that generated the current state belongs to {X, X', X2,} and X can be any of F, R, L, U, D, B.

A* algorithm

The A* method aims to find a solution close in length to the optimal one. The quality of the heuristic function determines the quality of its results. In this approach, at each step, the method selects the child node of the current node with the smallest cost. The cost function, $f(n)$, is computed by summing $g(n)$ and $h(n)$, where:

- n is the node;
- $g(n)$ represents the number of moves took so far;
- $h(n)$ is the heuristic function, computed as the sum of the minimum number of moves to bring each corner to its place, without considering the location of the other corners.

The heuristic function was created from scratch and tailored for the size of the cube.

To avoid cycles, A* follows the DFS rule for generating the allowed moves from a node.

Monte Carlo Tree Search (MCTS)

The MCTS approach we implemented is not a general solver, it aims to find a solution for the cube configuration received as a parameter.

This approach tries to find a solution in a maximum of 30 steps (which is equivalent to 30 moves). At each step, it runs the MCTS algorithm to find the best move to make from the current state, applies the determined rotation and adds it to the solution.

$$UCB(s_i) = v_i + C * \sqrt{\lg N/n_i}$$

v_i = Mean value of the node. This is the exploitation term.
C = A chosen constant.
N = Total number of simulations done for the parent node.
n_i = Number of simulations done for the current child node.

Figure 6 - UCB formula[1]

[1] https://builtin.com/machine-learning/monte-carlo-tree-search

The actual MCTS method runs over 700 iterations, each iteration consisting of four steps:

- Selection - starting from the root node (the initial cube state), it traverses the tree on the best children branch using the UCB formula (see Figure 6) until it reaches a terminal (the solved cube) or a not fully expanded node. In case of a not fully expanded node, it expands it; otherwise, it returns it.
- Expansion - for the not fully expanded node previously selected, it generates all its children, randomly chooses one of them, adds the child to the tree, and returns the child node.
- Simulation using epsilon-greedy strategy
  - To avoid cycles and to increase the chances of it finding a solution, we kept a list of visited states in the simulation;
  - Starting from the selected node (the terminal one from the selection phase or the child node returned by expansion), simulates a sequence of moves until it reaches the terminal state or the maximum depth (set as 20);
  - At each simulation step, it chooses a random move from the allowed ones in 20% of the cases, or the move with the smallest heuristic value (the heuristic function is the same as in A* approach) in 80% of the cases;
  - At the end of it, it returns a reward of $1 + \frac{1}{simulation\ depth}$, for the solved cube state at the end of the simulation, or $\frac{1}{1+h(current\ state)}$ otherwise.
- Backpropagation - traverses the tree backwards, starting from the selected node to the root, and updates the wins and scores for each node, according to the simulation results.

For the MCTS approach, the allowed moves are all half-turn metric rotations except for the inverses.

Q-learning

Just like the MCTS-based algorithm, Q-learning is not a general solver. In its center is the Q-table structure, a table that provides a value for a cube state, rotation made pair. Due to the large state space, the Q-table is initially empty, and states are added to the table as they are generated. A newly added record has the value set as zero for all 18 moves.

When calling the Q-learning method, the training process starts from scratch, so the results of the past runs are not considered in the current one and the Q-table is empty. Training the model consists of 50000 episodes. Each episode starts from the same initial cube state and runs until it reaches a terminal state (the solved cube) or the maximum depth (set as 30). An episode step consists of choosing a move using an epsilon-greedy policy (the move with the lowest heuristic value out of the allowed moves, determined as in MCTS, with a probability of $epsilon$, or a random move from the allowed ones with a probability of $1 - epsilon$), computing the reward and updating the current state and the Q-table accordingly.

The reward for the current state and the chosen move is computed as $heuristic\ before\ -\ heuristic\ after$, where $heuristic\ before$ is the heuristic value of the current state, and $heuristic\ after$ is the heuristic value of the new state, if the new state is non-terminal, or 100 otherwise.

The Q-table update follows the Temporal Difference formula. In addition to the epsilon-greedy strategy, we used epsilon decay. So, at each episode, epsilon is computed using the following formula: $\varepsilon = \varepsilon_{min} + (\varepsilon_{init} - \varepsilon_{min}) \cdot e^{-decayRate \cdot episode}$, where: ε_{min} is the minimum epsilon value (0.05), ε_{init} is the initial epsilon value (1), e is Euler's number, *decayRate* is the rate at which epsilon decreases, and *episode* is the number of the episode.

TEST DATA DESCRIPTION

The test data has been generated by code and consists of 45 data points, five for each optimal solution length from one to nine, which covers a wide range of cube state complexity. The optimal length for a cube configuration was determined by running the bidirectional BFS on that configuration.

The test data has been saved in a JSON file in which a record (see an example in Figure 7) contains: the test ID, the optimal solution length, and the resulting cube configuration as a 24-character string.

```
{
    "id": 13,
    "shuffleLength": 3,
    "shuffledCubeState": "UUUFRRDFFDFLDBDRLLBLBBUR"
},
```

Figure 7 - A record example from the test data file

EXPERIMENTS OVERVIEW

All the experiments were executed on a computer with the following characteristics: Windows 10 as operating system; an Intel® Core™ i5-8265U CPU @ 1.60GHz processor, with four physical cores / eight logical cores and a 1.80 GHz speed; Memory speed as 2400 MHz.

Testing Criteria

For each optimal solution length, we evaluate each implemented method considering four criteria, which cover the most important aspects of evaluation for such a project: First is the average execution time (measured in milliseconds). The second is average memory used (measured in kilobytes). The third is the success rate, and the last one is the average solution length.

Testing Method

In order to capture more accurate results, we run each algorithm four times on each test data, and we compute the average execution time, average memory used, success rate, and solution length. The average execution time and average memory used were calculated as the averages of the execution times and memory used, respectively, for the test data record. The success rate is calculated as the number of successes over the number of runs, and the average solution length is the average of the solution lengths determined in the successful runs. These results were saved in six JSON files, one for each algorithm.

The implemented code determined the execution time and memory used for a run. For execution time, we called the Start() method of a Stopwatch object before running the algorithm, then the Stop() method after the algorithm finished its execution. The time elapsed between these calls is the execution time. For memory usage, we called GC.GetTotalMemory() before and after running the algorithm, and the difference between these values divided by 1024 is the memory used, measured in KB.

Once these results have been determined, we grouped them based on the optimal solution length and computed the final results that will be used in comparing the algorithms. So, for an optimal solution length of n, we have:

- The average execution time is the average of all the average execution times of the test data of length n;
- The average memory used is the average of all the average memory used of the test data of length n;
- The success rate is the average of all the success rates of the test data of length n;
- The average solution length is computed as $\frac{\sum_{i=1}^{4} (S_i \cdot l_i)}{\sum_{i=1}^{4} S_i}$, where S_i is the success rate for test data i and l_i is the average solution length for test data i.

The final results were also saved in six JSON files, one for each algorithm. All the results and the test data are available on GitHub [13].

EXPERIMENTAL RESULTS

In this section, we present the final results of our experiments, considering all four criteria.

Average Execution Time

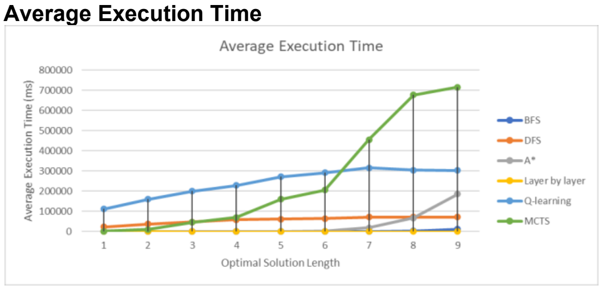


Figure 8 - Average Execution Time Diagram

Considering the information provided by Figure 8, we can draw the following conclusions about the average execution time:

- Layer-by-layer has the smallest execution time, its value stabilizing somewhere around 6ms, so its execution time is almost constant during experiments;

- Bigger than Layer-by-layer average execution time, the average execution time for bidirectional BFS is smaller than the others for all optimal solution lengths;
- For A* and bidirectional BFS, the average execution time grows with the increase of optimal solution length, which is to be expected considering the exponential growth of the state space;
- DFS has a somewhat constant average execution time during experiments. For the longer solutions (five moves or more), where it tends to find a solution rarely, the average execution time of DFS reaches its maximum value;
- MCTS and Q-learning have the most significant average execution time overall, which is caused by the training period.
- For lengths of one to six, Q-learning has the most significant average execution time, and for lengths of seven to nine (where it rarely finds a solution), it is surpassed only by MCTS;
- The average execution time of MCTS grows dramatically from one length to another, especially for the longer optimal solutions (six to nine moves), which is to be expected considering the way MCTS approach searches for a solution (a more complex cube configuration needs more moves to be solved, so it needs running MCTS algorithm several times).

Average Memory Used

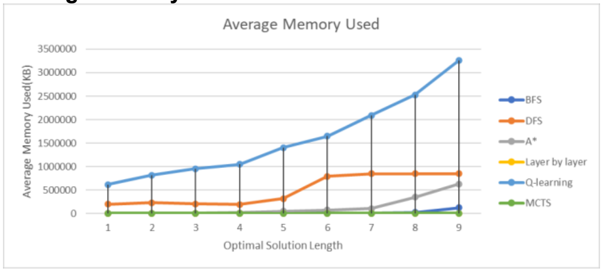


Figure 9 - Average Memory Used Diagram

Considering the information provided by Figure 9, we can draw the following conclusions about the average memory used:

- Layer-by-layer doesn’t use a big amount of memory, its value is zero or close to zero for each optimal solution length;
- In general, bidirectional BFS consumes a pretty small amount of memory compared to other algorithms;
- The average amount of memory used by bidirectional BFS and A* increases with the optimal solution length, which is to be expected considering the exponential growth of the state space;
- The average amount of memory used by DFS is large for all optimal solution lengths and, for the longer ones, where it rarely or never finds a solution (six to nine moves), the average amount of memory used stabilizes around 846000 KB;
- MCTS approach uses a small and almost constant amount of memory during testing (2000 - 3000 KB), which is due to the small tree that is created and then destroyed at each step of the approach;
- Q-learning consumes the biggest amount of memory, its values increasing dramatically from one optimal solution length to another, which is to be expected due to the increasing size of Q-table.

Average Solution Length

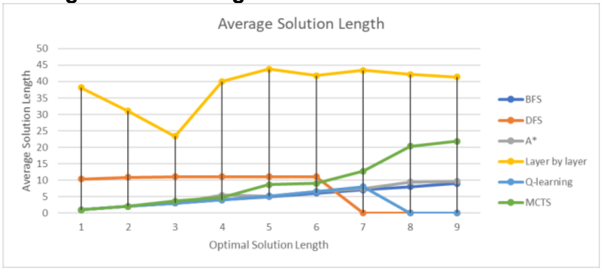


Figure 10 - Average Solution Length Diagram

Considering the information provided by Figure 10 (where a value of zero means that the algorithm failed in finding a solution, we can draw the following conclusions about the average solution length:

- Layer-by-layer always finds the longest solutions, and the cube state complexity does not influence the solution length found by it, but rather the way the cube state lends itself to the steps of the method.
- Due to the way DFS traverses the tree, it tends to find a solution close in length to its maximum depth (i.e. 11), even for less complex cube configurations, and, for the longer optimal solutions (seven to nine moves), it never finds a solution;
- A* determines a solution close in length to the optimal one, which shows that the heuristic is good enough.
- Bidirectional BFS always finds the optimal solution;
- For the successful runs, Q-learning tends to find a solution close in length to the optimal one, but it fails to discover a solution for complex cube states (eight and nine moves);
- The MCTS approach tends to find long solutions, and for the cubes that can be solved in eight or nine moves, the increase in solution length is dramatic.

Success Rate

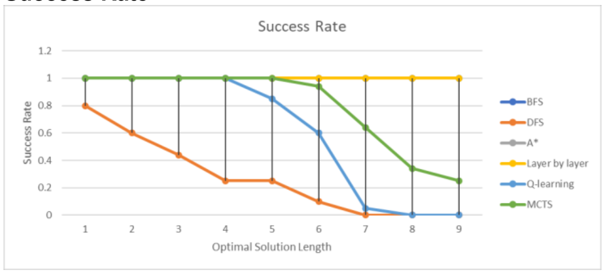


Figure 11 - Success Rate Diagram

Considering the information provided by Figure 11, we can draw the following conclusions about the success rate:

- Because we didn't set any limitations, A*, Layer-by-layer and bidirectional BFS always find a solution, so their success rate is 1;
- DFS does not always succeed to find a solution even for the smallest optimal solution lengths. Due to the way it traverses the tree and chooses a move at each step, the success rate of DFS is not entirely dependent on the optimal solution length.
- Q-learning has always found a solution for the optimal solution lengths of one to four moves, but its efficiency decreases as the optimal solution length increases and for the tests of eight or nine moves in solution, its success rate is zero.
- The success rate of the MCTS approach also decreases as the optimal solution length increases, but, in contrast to Q-learning, it is never zero.

CONCLUSIONS

The project presented in this paper successfully balances a user-friendly interface with the functionalities of six different methods for solving the pocket Rubik's Cube. The experimental test data provides a varied and robust set of cube configurations that allows us to observe and compare the efficiency of the algorithms we developed. Based on the experimental results, we can conclude that all implemented methods have their advantages and disadvantages, and in order to choose the best one of them, we should focus on the user's requirements. Thus, DFS is not a good approach overall. Bidirectional BFS always finds the optimal solution in a decent amount of time and memory. Layer-by-layer is the quickest and uses the smallest amount of memory, but the solution found by it is longer compared to the other methods. A* is ideal in terms of success rate and provides a solution of close to optimal length, but is not as efficient in terms of memory and execution time as bidirectional BFS or Layer-by-layer. The Q-learning approach we developed is not the ideal approach for solving the Rubik's cube, because of its success rate for more complex cube states, memory, and execution time . The MCTS approach is not ideal either, because of its execution time, success rate, or solution length, especially for the complex cube configurations.

In future work in this direction, we aim to focus on machine learning methods, as a more effective training approach can significantly improve them. Moreover, we want to turn our algorithms into general solvers. Therefore, instead of aiming to solve only the current cube configuration, we want to run each algorithm on all possible configurations and save the method's results in an external file or database. When the user presses one of the solving buttons, the algorithm results should be retrieved directly from this source.

REFERENCES

1. Steinparz, C. A., Hinterreiter, A. P., Stitz, H., & Streit, M. (2019). Visualization of Rubik's Cube Solution Algorithms. In EuroVA@ EuroVis (pp. 19-23).
2. Jeevan, J., & Nair, M. S. (2022). On the performance analysis of solving the Rubik's cube using swarm intelligence algorithms. Applied Artificial Intelligence, 36(1), 2138129..
3. McAleer, S., Agostinelli, F., Shmakov, A., & Baldi, P. (2018). Solving the Rubik's cube without human knowledge. arXiv preprint arXiv:1805.07470.
4. Agostinelli, F., McAleer, S., Shmakov, A., & Baldi, P. (2019). Solving the Rubik's cube with deep reinforcement learning and search. Nature Machine Intelligence, 1(8), 356-363..
5. Demaine, E. D., Demaine, M. L., Eisenstat, S., Lubiw, A., & Winslow, A. (2011). 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 Berlin Heidelberg.
6. Demaine, E. D., Eisenstat, S., & Rudoy, M. (2017). Solving the Rubik's Cube Optimally is NP-complete. arXiv preprint arXiv:1706.06708.
7. Higo, R., Yamakawa, Y., Senoo, T., & Ishikawa, M. (2018, October). Rubik's cube handling using a high-speed multi-fingered hand and a high-speed vision system. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 6609-6614). IEEE.
8. Zeng, D. X., Li, M., Wang, J. J., Hou, Y. L., Lu, W. J., & Huang, Z. (2018). Overview of Rubik's cube and reflections on its application in mechanism. Chinese Journal of Mechanical Engineering, 31, 1-12.
9. Kunkle, D., & Cooperman, G. (2008). Solving rubik's cube: disk is the new ram. Communications of the ACM, 51(4), 31-33.
10. Johnson, C. G. (2021). Solving the Rubik's cube with stepwise deep learning. *Expert Systems*, *38*(3), e12665.
11. Chakraborty, A., & Kar, A. K. (2017). Swarm intelligence: A review of algorithms. Nature-inspired computing and optimization: Theory and applications, 475-494.
12. Wikipedia, Rubik's Cube, n.d., available online at: https://en.wikipedia.org/wiki/Rubik%27s_Cube
13. https://github.com/AncaNicu/Algorithms-for-solving-2x2-Rubik-s-cube