Rubik's Cube Solver 3x3x3

CS644 Final Project - Fall 2020

Divyesh Jagetia

A 3x3x3 Rubik's cube is a Stochastic non deterministic puzzle with absolutely enormous possibilities of orientations. This project covers all the Artificial Intelligence Optimization methods that can be used and shows the most optimal method with application. It starts from scanning a cube using computer vision, to representing it in code and solving it using combinatorial optimization method and group theory with Iterative deepening A heuristic search in phases. This project report also highlights few of the issues with other methods like Deep Reinforcement learning.*

Background

To solve a puzzle, it is important to first understand the puzzle. A Rubik's cube is a popular puzzle that became popular in the 1980s. Ernõ Rubik, a professor from Hungary, invented the Rubik's Cube in 1974 to help his students better understand three-dimensional problems. It took the inventor a month to figure out how to solve the magic cube.

Now the world record for fastest time solving a Rubik's cube is 3.47 seconds, set in 2018 by Yusheng Du. The puzzle offers a great challenge. This is because there are approximately forty-three quintillion possible states the cube can be in, and only one correct solution. This number is obtained by knowing that 8 corner cuboids can be rotated in three different ways, and 12 edge cuboids can be rotated in 2 different ways. Knowing this, we can multiply the number of combinations of the corners by the number of combinations of the edges and divide by the number of correct orbits. This eliminates any of the impossible cube states.

$$\frac{(3^8 * 8!)(2^{12} * 12!)}{3 * 2 * 2} \approx 4.3 * 10^{19}$$

Notation: A Rubik's cube has 6 faces. If you hold a cube flat in your hand, the face that is on top is known as the up face, the opposite of this face is the down face. The face in front of you is known as the front face, and the one opposite of this is known as the back face. The last faces are the left and the right face, and as the name suggests, they are the faces on the left and right. Figure 1 better illustrates the orientation of the faces.

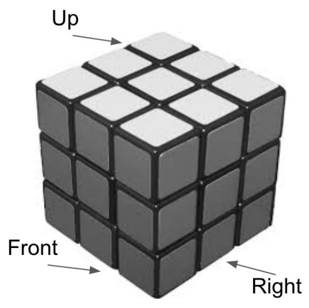
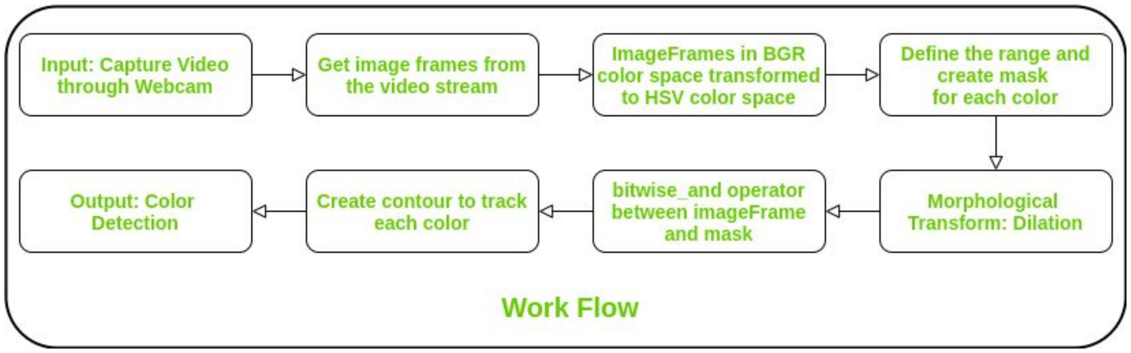
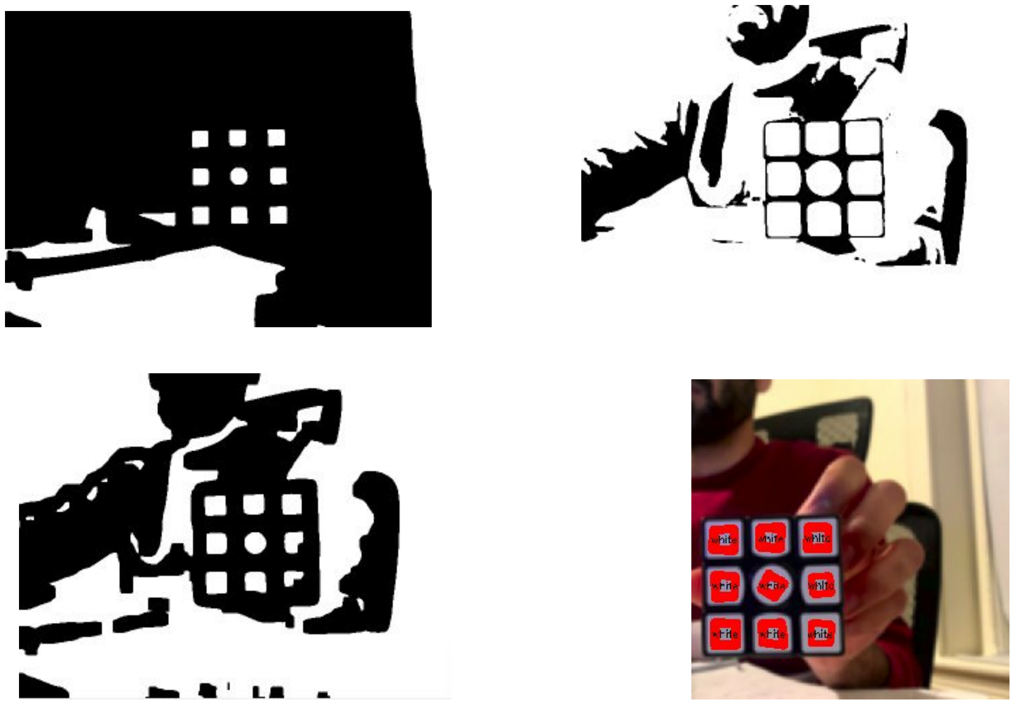


Figure 1. Face Notation

Computer Vision - Scanning

Step 1: Input: Capture video through webcam.
Step 2: Read the video stream in image frames.
Step 3: Convert the imageFrame in BGR(RGB color space represented as three matrices of red, green and blue with integer values from 0 to 255) to HSV(hue-saturation-value) color space. *Hue* describes a color in terms of *saturation*, represents the amount of gray color in that color and *value* describes the brightness or intensity of the color. This can be represented as three matrices in the range of 0-179, 0-255 and 0-255 respectively.
Step 4: Define the range of each color and create the corresponding mask.
Step 5: Morphological Transform: Dilation, to remove noises from the images.
Step 6: bitwise_and between the image frame and mask is performed to specifically detect that particular color and discard others.
Step 7: Create contour for the individual colors to display the detected colored region distinguishably.
Step 8: Output: Detection of the colors in real-time.


Input: Capture Video through Webcam
Get image frames from the video stream
ImageFrames in BGR color space transformed to HSV color space
Define the range and create mask for each color
Morphological Transform: Dilation
bitwise_and operator between imageFrame and mask
Create contour to track each color
Output: Color Detection
Work Flow



(Top Left) White Mask, (Top right) Blak Mask, (Bottom Left) Contour Mapping, (Bottom right) Result

Representing the Cube in Code

There are many different ways a Rubik’s cube can be represented in code, each comes with perks with regards to space complexity and ability to be easily manipulated by a solving algorithm. A few common ways of representing a cube that I explored are: one-dimensional array, two-dimensional array, three-dimensional array, object-oriented programming, matrix, and binary. Additionally, there are many more ways we can represent the cube such as the use of strings and other data structures. Another important design choice to consider is how to represent the Facelet of the cube. Usually, a Facelet will store the color, but optionally it might also store the location depending on how the cube is being represented.

One-dimensional array: Using a one-dimensional array each Facelet of the cube can be labeled 0 through 53. Each of these labels maps the respective Facelet to the index in the array. A move to the cube can be executed by performing a mapping function that rearranges the Facelets accordingly.

A cube can be represented using a two-dimensional array to store an array of faces. This approach stores six faces, where each face is an array of the nine Facelets corresponding to the given face.

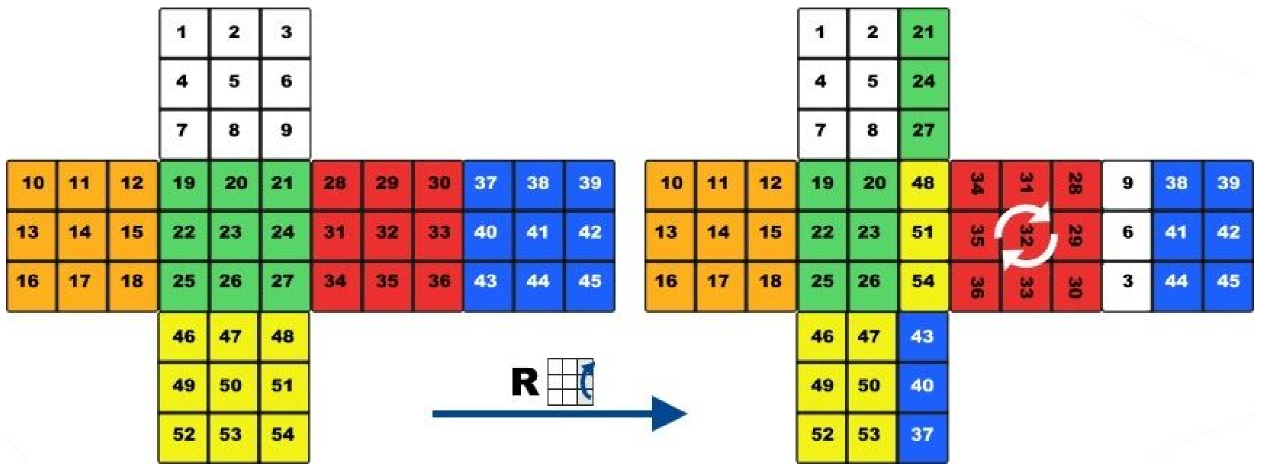
```
              +-------------+
              |  1   2   3  |
              |             |
              |  4   u   5  |
              |             |
              |  6   7   8  |
+-------------+-------------+-------------+-------------+
|  9  10  11  | 17  18  19  | 25  26  27  | 33  34  35  |
|             |             |             |             |
| 12   l  13  | 20   f  21  | 28   r  29  | 36   b  37  |
|             |             |             |             |
| 14  15  16  | 22  23  24  | 30  31  32  | 38  39  40  |
+-------------+-------------+-------------+-------------+
              | 41  42  43  |
              |             |
              | 44   d  45  |
              |             |
              | 46  47  48  |
              +-------------+
```

Three-Dimensional Array: Using object-oriented programming(OOP) principles a cube was represented using a three-dimensional array. Working with an abstract cubie class, corners, edges, and centers can all extend from this class. The three-dimensional array consists of cubies. Each one located in its corresponding location to where its three-dimensional coordinate would be on a real cube.

Matrix: A matrix was used to store the cubes state by using a 6 by 54 matrix. Each of the columns represents one Facelet on the cube and a one is placed in the column corresponding to its color row. An example of an encoded matrix can be seen below

$$\text{6 Colors}\left\{\overbrace{\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \ldots \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \ldots \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & \ldots \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \end{pmatrix}}^{\text{54 Facelets}}\right.$$

Defining and storing constraints: Considering all the maneuvers, lookup tables were created to store the constraints. For eg, right side move would result in the array represented in the picture below. Note: This shows the need of group theory.



Current Algorithms and methods:

Fredich's method:

The most common methods are the beginner's method and CFOP. The beginner's method consists of 7 steps: cross, corners, middle layer, top cross, fix cross, permutation of corners, and orientation of corners. In contrast, the more advanced method CFOP is only 4 steps: cross, first two layers(F2L), the orientation of the last layer(OLL), and the permutation of the last layer(PLL). Orientation refers to twisting and flipping a cuboid in its current location, while permutation refers to rearranging the cuboids to a new location

The optimal number of moves to solve a Rubik's cube is 20. This was discovered in 2010 by Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. They were able to achieve this by reducing the 43 quintillion possible combinations by a factor of 48 using symmetry and mirrored states. There are 24 ways you can orient a Rubik's cube in space and you can multiply this by two to account for mirrored states. After this reduction, only a set of 55,882,296 cube states had to be examined. Using Google's computers they were able to generate the numbers shown in Table 2. Calculating these numbers would take a good desktop computer 35 CPU-years.

Table 2: Scramble Distance Possible Positions

Distance	Total Possible Positions
0	1
1	18
2	243
3	3,240
4	43,239
5	574,908
6	7,618,438
7	100,803,036
8	1,332,343,288
9	17,596,479,795
10	232,248,063,316
11	3,063,288,809,012
12	40,374,425,656,248
13	531,653,418,284,628
14	6,989,320,578,825,350
15	91,365,146,187,124,300
16	$\sim$ 1,100,000,000,000,000,000

Neural Networks:

Trying to solve a Rubik's cube using a neural network tends to be a difficult challenge since there is a large state space with only one correct solution. This means the neural network will have a tough time knowing how to update the states since randomly doing moves will not likely stumble upon the solved cube state. One way to improve the performance of a neural network is by the use of boosting using weak learners. One group tested a program using this method and found "AdaBoost had worse classification accuracy than a baseline [Stochastic Gradient Descent] method that did no boosting." Their findings concluded that neural networks are expensive to train and boosting did not work well because weak learners are better suited when the weak learner is truly a weak learner.

Neural networks can be combined with other algorithms to make a powerful solver. This is exactly what powers the DeepCubeA solver. DeepCubeA "works by using approximate value iteration to train a deep neural network (DNN) to approximate a function that outputs the cost to reach the goal (also known as the cost-to-go function)." This DNN is then combined with an A* search algorithm to help approximate and compare the values of various cube states while searching for a solution.

Another group used a neural network to help improve their solving algorithm by swapping between two different algorithms. They called their method Approximate Policy Iteration(API). The API "algorithms iterate between two policies: a slow policy (tree search), and a fast policy (a neural network)." By doing this they were able to use the perks of both a search tree(Monte Carlo Search Tree) and a neural network to train and build an accurate model that could efficiently solve Rubik's cubes.

Another option could be to give the neural network easier training data in the beginning. An example of this would be to give it cubes that have only been scrambled with 3 moves instead of 1000 moves. This will allow the neural network to be trained at a much faster rate. As the neural network starts to become more proficient at solving cubes in an easier state, we can increase the difficulty by training on cubes that have more scramble moves applied. This also helps overcome the problem of there being a sparse reward in a large state space. But these models tend to take a long time to train however they find the optimal solution to solving the cube.

Brute Force Search- A* Method:

A* is a popular heuristic search algorithm that seeks to find the smallest cost or shortest path to a given goal node, in this case, the solved cube. This is done by using the following equation:

$$f(n) = g(n) + h(n)$$

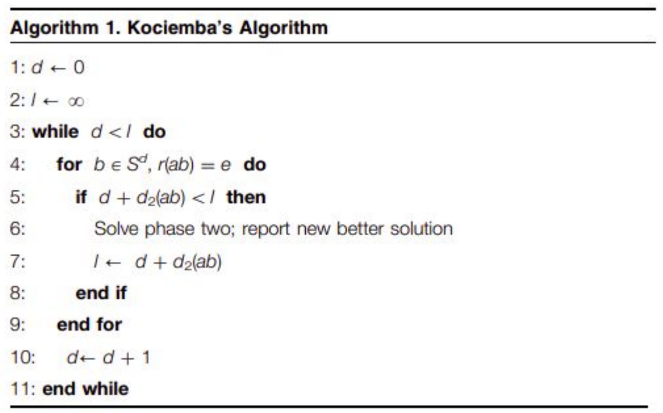
This function helps the search algorithm determine which node to advance to. This is applied to all the available nodes from the given starting node. N is the next node on the path, g(n) is the cost of the path from the start node, and h(n) is a heuristic that estimates the cost of the cheapest path from n to the goal. The algorithm chooses to advance to the node that produces the lowest f(n). Also important to note if it is not possible to extend the path, or it has reached some terminating state then it goes back to the starting node and chooses the next minimum f(n) value. This algorithm is memory intensive since it has to store all the nodes it uses in memory. Similar to the neural network, it works well if combined with another algorithm technique.

Korfs/Iterative-Deepening-A*

Korf's algorithm found an optimal solution to solve a Rubik's cube. The algorithm uses "iterative-deepening-A* (IDA*), with a lower bound heuristic function based on large memory-based lookup tables, or pattern databases."[9] This method uses the A* algorithm previously discussed but it can improve the searching with the pattern databases. These pattern databases "store the exact number of moves required to solve various subgoals of the problem." This technique allows this algorithm to solve the cube in the optimal number of moves, but this comes with the trade-off of time. The DeepCube team concluded in their comparison that "Korf's algorithm will always find the optimal solution from any given starting state; but, since it is a heuristic tree search, it will often have to explore many different states and it will take a long time to compute a solution." Since this algorithm directly scales with the amount of memory, as computers harness more computing power and memory increases, this algorithm will see a drastic increase in time. Improving this algorithm with better indexing techniques was required. Additionally finding better ways to hash the cube would help minimize the amount of memory required to store all the cube states in the pattern database.

Kociemba/2-Phase Algorithm Combinatorial Optimization method:

Finally, this algorithm merges the above methods optimally, i.e. solves all the issues of previous methods. As the name implies, this algorithm has two phases. The first phase puts pieces into the correct position but does not change the orientation of any of the corners or edges. This is done by restricting the moves to only a small set of moves (U, D, B2 F2, L2, R2). Once this phase finds "the shortest number of moves required to do this, it continues until it has a large enough supply of possible solutions ranging from the lowest number of moves to a much larger number."[10] Advancing on to the second phase once all the permutations are restored, the algorithm searches for an optimal solution. This is not an optimal solution of the second phase, but rather an optimal solution of combining the moves of phase 1. For example, an 8 move Phase 1 followed by a 15 move Phase 2 is less optimal than a 10 move Phase 1 followed by a 5 move Phase 2. Because the two phases break the problem into smaller problems, multiple solutions can be found very quickly for each step and combined to make the most favorable solution. This algorithm relies on human domain knowledge of the group theory of the Rubik's Cube. Kociemba's algo always solves any cube given to it, and it runs very quickly. Kociemba solved each cube in under a second. This algorithm is extremely fast and is one of the best algorithms used to quickly solve the cube.



Algorithm 1. Kociemba's Algorithm

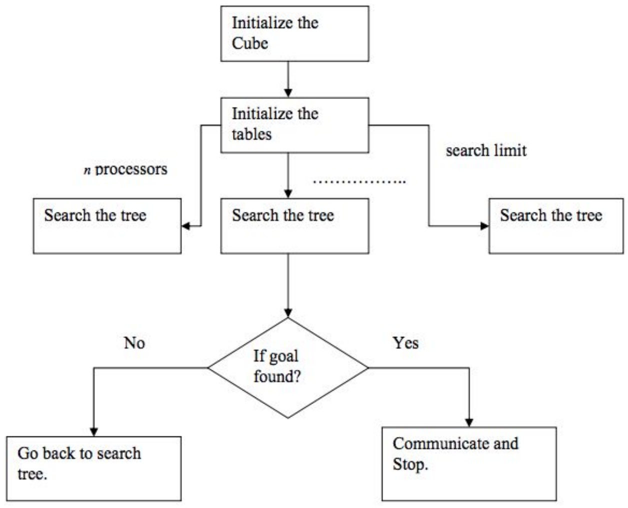
```
1: $d \leftarrow 0$
2: $l \leftarrow \infty$
3: while $d < l$ do
4:     for $b \in S^d, r(ab) = e$ do
5:         if $d + d_2(ab) < l$ then
6:             Solve phase two; report new better solution
7:             $l \leftarrow d + d_2(ab)$
8:         end if
9:     end for
10:    $d \leftarrow d + 1$
11: end while
```

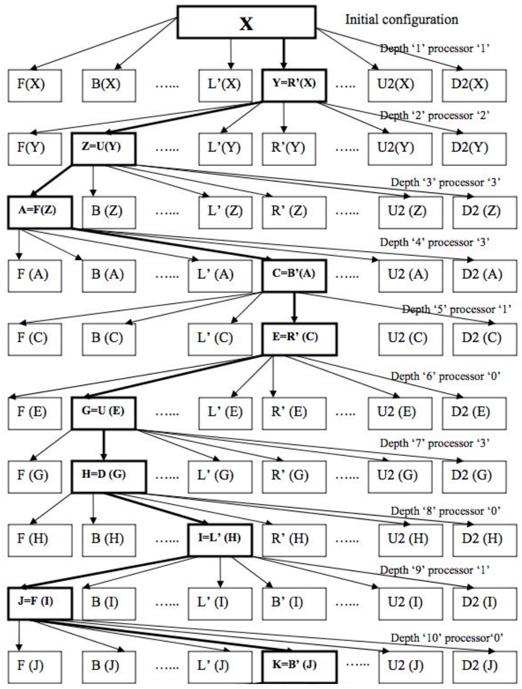
.

The subset H is composed of all positions that have the following characteristics:
1. All corners and edges are in their default orientation (as defined earlier).
2. The edge cubies that belong in the middle layer are located in the middle layer.

Kociemba's algorithm takes the original position a, compute r(a), the relabeling; solve the relabeled puzzle with some sequence b ∈ subset S*; apply those moves to an original cube yielding ab which lies in H, and then finish the solution with another sequence c ∈ A* such that abc is the solved cube. The final solution sequence is bc.

In algorithm 1, d_2 is a table lookup that takes a position in H and returns the distance to the identity element (e) using moves in A. Kociemba actually uses a smaller, faster table that gives a bound on this value. The for loop is implemented by a depth-first recursive routine that maintains ab incrementally and has a number of further refinements, such as not permitting b to end in a move in A. The phase-two solution process is omitted both because it is straightforward and because it takes much less time than enumerating phase-one solutions. Below is a visualisation and workflow of how iterative deepening A* search is conducted.




X
Initial configuration
Depth '1' processor '1'
F(X) B(X) L'(X) Y=R'(X) U2(X) D2(X)
Depth '2' processor '2'
F(Y) Z=U(Y) L'(Y) R'(Y) U2(Y) D2(Y)
Depth '3' processor '3'
A=F(Z) B (Z) L' (Z) R' (Z) U2 (Z) D2 (Z)
Depth '4' processor '3'
F (A) B (A) L' (A) C=B'(A) U2 (A) D2 (A)
Depth '5' processor '1'
F (C) B (C) L' (C) E=R' (C) U2 (C) D2 (C)
Depth '6' processor '0'
F (E) G=U (E) L' (E) R' (E) U2 (E) D2 (E)
Depth '7' processor '3'
F (G) H=D (G) L' (G) R' (G) U2 (G) D2 (G)
Depth '8' processor '0'
F (H) B (H) I=L' (H) R' (H) U2 (H) D2 (H)
Depth '9' processor '1'
J=F (I) B (I) L' (I) B' (I) U2 (I) D2 (I)
Depth '10' processor'0'
F (J) B (J) L' (J) K=B' (J) U2 (J) D2 (J)

2phase limitations - Depth <11. It does not use phase-2 maneuvers with length >=10.

Code:

Github Repository for this Project

Running demo video

Future Scope:

Beyond the 3x3, other cube sizes such as the 2x2 and 4x4 cubes on up to NxN cube can be solved using the methods discussed. Additionally, other combinatorial puzzles or group theory problems can benefit from this method as well. After making proper adjustments, other puzzles can be solved in the same way. Puzzles such as the 15 Sliding Puzzle and Lights Out can be solved using similar approaches or in some cases even using the same model. Applying Reinforcement Learning by building states, actions and reward policy could make the agent learn everything on its own. Even though it is time consuming, we might find a state-of-art-performance for the most optimal solution.

Conclusion:

There are many approaches to solve a Rubik's cube. All are correct. Depending on what design choice is valued the most will determine the method selected. If the goal is an easy algorithm to develop, then modeling the way humans solve the cube using a layer-by-layer approach or CFOP is the best solution. If having the optimal number of moves is the desired result, then implementing Korf's Iterative-Deepening-A*algorithm will suffice. If time is a concern, then a quicker algorithm such as Kociemba's 2-Phase algorithm will yield a much faster solver. Developing solutions to hard combinatorial puzzles such as the Rubik's cube helps in moving forward with difficult engineering challenges or problems that are too hard to solve using an exhaustive brute force approach.

References:

[1]: Rubik's About Us. Our Heritage. https://www.rubiks.com/en-us/about

[2]: World Cubing Association. (January 1, 2020). WCA Regulations. https://www.worldcubeassociation.org/regulations/

[3]: Guinness World Records. (November 24, 2018). Fastest time to solve a Rubik's Cube. https://www.worldcubeassociation.org/regulations/

[4]: Cube 20. (July 1, 2010). God's Number is 20. https://cube20.org/

[5]: Irpan, Alexander. (2016). Exploring Boosted Neural Nets for Rubik's Cube Solving. https://www.alexirpan.com/public/research/nips_2016.pdf

[6]: McAleer, Stephen, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. (2018). Solving the Rubik's Cube with Approximate Policy Iteration.

[7]: Agostinelli, F., McAleer, S., Shmakov, A. et al. (2019). Solving the Rubik's cube with deep reinforcement learning and search. https://doi.org/10.1038/s42256-019-0070-z

[8]: Ben Botto. (May 26, 2019) Sequentially Indexing Permutations: A Linear Algorithm for Computing Lexicographic Rank. https://medium.com/@benjamin.botto/sequentially-indexing-permutations-a-linear-algorithm-for-computin g-lexicographic-rank-a22220ffd6e3

[9]: Korf, Richard E. (1997). Finding optimal solutions to Rubik's cube using pattern databases. https://www.aaai.org/Papers/AAAI/1997/AAAI97-109.pdf

[10]: Ruwix. Herbert Kociemba's Optimal Cube Solver — Cube Explorer. https://ruwix.com/the-rubiks-cube/herbert-kociemba-optimal-cube-solver-cube-explorer/