Optimizing Rubik's Cube Game Using State Space Search

Shreya Manyam[1,2], Vibhansh Goel[2], Battu Ujwal Reddy[2] and Erramshetti Abhilash[2]

[1] Group Name: Agile
[1] IIT Vadodara, ICD

Abstract

This project focuses on developing an AI-based Rubik's Cube solver by using state space search techniques. The Rubik's Cube presents a complex combinatorial type of problem with a vast search space, making it an ideal candidate for heuristic optimization methods. The solution process is broken down into three key stages: solving the first layer (cross, face, and layer), completing the second layer (placing correct edge piece of second layer), and solving the third layer (identifying patterns, permuting edge and corners pieces) by Rubik's layer solving approach. For each stage, a series of state transitions and predefined moves are employed to navigate towards the goal state of a fully solved cube. Heuristic functions like the Manhattan and Hamming distances guide the search, enabling efficient move selection while minimizing unnecessary transitions. By structuring the problem as a state space search, the solver is capable of optimizing both performance and move efficiency, offering insights into the broader application of AI in strategic games and puzzles.

Keywords: Heuristic Optimization, State Space Search, Rubik's Cube, Artificial Intelligence (AI), Puzzle Solving Algorithms, Rubik's layer approach.

I. Introduction

State space search is a powerful technique in artificial intelligence (AI), widely used for solving complex problems by exploring all possible states of a given system. In strategy games, AI opponents must analyze a vast number of possible moves in order to make it's decisions. The challenge lies in optimizing the search process to reduce computational cost while ensuring effective decision-making. By utilizing state space search, combined with heuristics, pruning techniques, and machine learning, AI opponents can dynamically adapt to in-game scenarios, improving their strategy over time. This project focuses on optimizing state space search for AI bots in strategy games like Rubix Cube, aiming to balance between decision quality and computational efficiency, guiding the challenger towards the goal.[1].

II. Problem Statement

The Rubik's Cube presents a challenging combinatorial puzzle, where the objective is to return a scrambled cube to its solved state, with each of the six faces consisting of a uniform color. Given the complexity of the puzzle, with over 43 quintillion possible configurations, finding an efficient solution is non-trivial. The primary challenge lies in identifying a series of moves that will transition the cube from its initial scrambled state to the solved state in the fewest possible steps.

Traditional methods of solving the Rubik's Cube often rely on memorized algorithms or brute force approaches, but these are inefficient and computationally expensive for large-scale or automated solutions. This project focuses on utilizing state space search techniques combined with heuristic optimization to improve the efficiency of solving the cube. [2].

III. State Space Approach

The state space search for solving the Rubik's Cube involves:

- **Defining the State Space:** Representing each configuration of the cube as a state. This includes 43 quintillion possible cube states, but only a fraction of these need to be explored due to optimizations.
- **Action Set:** Defining all possible moves (rotations) that can transition the cube from one state to another.
- **Initial State:** The scrambled cube.
- **Goal State:** The fully solved cube.
- **Exploration Strategy:** Depending on the step, different search techniques like BFS (Breadth-First Search), DFS (Depth-First Search), or heuristic-based searches (A*) are

used to explore the state space efficiently. These techniques help guide the cube closer to the goal by exploring possible moves, evaluating their outcomes, and eliminating inefficient paths.

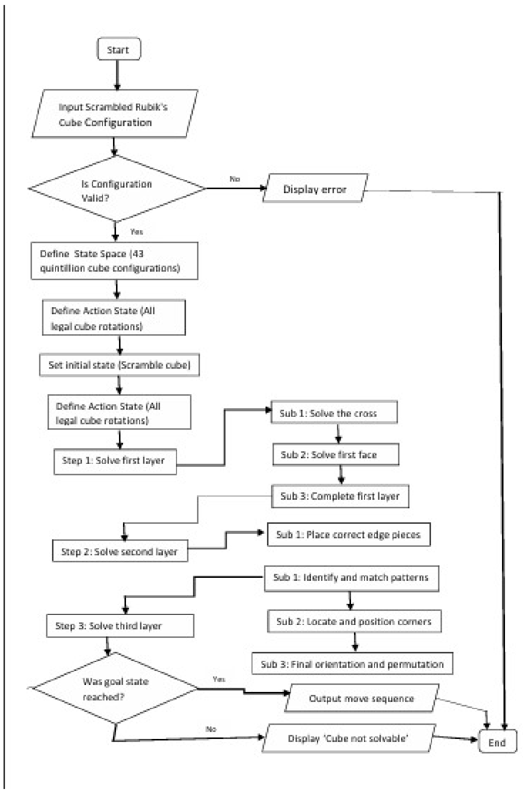
Start
Input Scrambled Rubik's Cube Configuration
Is Configuration Valid?
No
Display error
Yes
Define State Space (43 quintillion cube configurations)
Define Action State (All legal cube rotations)
Set initial state (Scramble cube)
Define Action State (All legal cube rotations)
Step 1: Solve first layer
Sub 1: Solve the cross
Sub 2: Solve first face
Sub 3: Complete first layer
Step 2: Solve second layer
Sub 1: Place correct edge pieces
Sub 1: Identify and match patterns
Step 3: Solve third layer
Sub 2: Locate and position corners
Sub 3: Final orientation and permutation
Was goal state reached?
Yes
Output move sequence
No
Display 'Cube not solvable'
End

Figure 1. Flowchart for Rubik's Cube State Space Search

IV. Algorithm

A. Step 1: Solving Layer 1 (Cross, Face, and Layer)

1 *Sub 1: Solving the Cross*

- **State Space:** Identify the positions of all edge pieces that contain the color of the chosen face (usually white). Each edge piece corresponds to a state.
- **Action Set:** Rotate the cube and perform necessary moves to position the four edge pieces around the center of the face, forming a cross. Use legal cube rotations to preserve existing edges.
- **Goal State:** Achieve the cross where the edges are correctly aligned with adjacent face's center pieces.

2 *Sub 2: Solving the First Face*

- **State Space:** Once the cross is complete, locate the corner pieces that have the first/front face color..

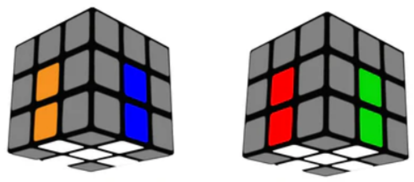

Figure 2. White cross solved

- **Action Set:** Use algorithms to position the corner pieces correctly.
- **Goal State:** All four corner pieces should be correctly positioned and oriented, forming the complete first face.

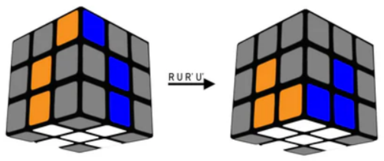

Figure 3. White face solved

3 *Sub 3: Completing the First Layer*

- **State Space:** Check the alignment of the corner pieces with the adjacent edges on the side faces.
- **Action Set:** Perform specific moves to adjust the orientation of corner pieces while maintaining the integrity of the solved cross and face.
- **Goal State:** The first layer is fully solved, with all corner pieces aligned and matching the adjacent side colors.

B. Step 2: Solving the Second Layer

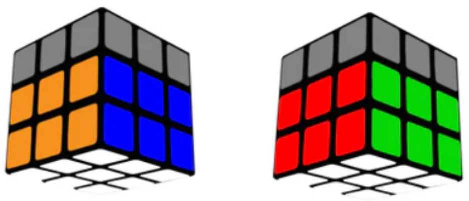

Figure 4. Second layer solved

1 *Sub 1: Placing correct edge Pieces*

- **State Space:** Locate the edge pieces that belong in the second layer. These pieces do not contain the color of the first face.

- **Action Set:** Match these edge pieces with the center color of the adjacent side face .
- **Goal State:** The edge piece is positioned between the two correct centers.

C. Step 3: Solving the Third Layer

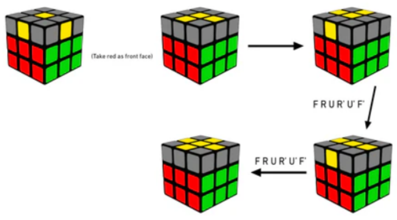

Figure 5. Third layer solving steps

1 Sub 1: Identify and Match Patterns

- **State Space:** Determine the state of the third layer by identifying the orientation and position of the edges peices (L-shape, line, or cross).
- **Action Set:** Use algorithms according to pattern to orient edges and make a cross(if cross is present earlier this part can be skipped).
- **Goal State:** Form a cross on the third layer without disrupting the solved lower layers.

2 Sub 2: Locating and Positioning Corners

- **State Space:** Check the position and orientation of the corner pieces on the third layer.
- **Action Set:** Use algorithms to swap corners and correctly place them.
- **Goal State:** All corners are in their correct positions, but may still need orientation.

3 Sub 3: Final Orientation and Permutation

- **State Space:** Once corners and edges are positioned, focus on orienting the pieces (solving parity issues if any).
- **Action Set:** Use set algorithms to finish the final orientation.
- **Goal State:** The third layer is fully solved, completing the cube.

V. Key Components of the Rubik's Cube Solver

A. 1. Cube Representation

The Rubik's Cube is represented as a dictionary of six faces (Up, Down, Front, Back, Left, Right). Each face is a 3x3 grid of colors.

- **Up (U):** Top face
- **Down (D):** Bottom face
- **Front (F):** Front face
- **Back (B):** Back face
- **Left (L):** Left face
- **Right (R):** Right face

B. 2. Cube Rotation Functions

The rotation of faces (90 degrees clockwise and counterclockwise) is fundamental for manipulating the cube. The function definitions include:

- **rotate_face_clockwise:** Rotates a face of the cube 90 degrees clockwise.
- **rotate_face_counter_clockwise:** Rotates a face of the cube 90 degrees counterclockwise.

Each move (U, D, F, B, L, R) and its inverse (U′, D′, etc.) affects the face and its adjacent edges.

C. 3. Breadth-First Search (BFS) Algorithm

BFS explores the cube state space layer by layer, applying all possible moves to each state. This section outlines the BFS process:

- Converts the cube into a tuple for easy comparison.
- Traverses through possible states by applying a sequence of moves.
- Returns the sequence of moves required to solve the cube or the intermediate states if a solution isn't found.

D. Heuristic Function for A*

In the context of the A* search algorithm, the **heuristic function** plays a critical role in determining the efficiency of the search. A good heuristic can significantly reduce the number of states the algorithm explores, leading to a faster solution.

1 What is a Heuristic Function?

A heuristic function, denoted as $h(n)$, is used to estimate the cost from the current state n to the goal state in a search problem. In the case of the Rubik's Cube, the goal is to have all the pieces in their correct positions with matching colors on each face.

2 Manhattan Distance Heuristic

One commonly used heuristic for the Rubik's Cube is the **Manhattan Distance**, which is an admissible and often effective choice. The Manhattan Distance measures how far each cubelet (individual piece) is from its correct position, considering the number of moves needed to place it correctly. It only takes into account moves along the cube's edges and doesn't consider rotations or the constraints imposed by solving the cube as a whole.

For each cubelet, the heuristic computes the number of moves required to bring it into the correct position, assuming it could move independently of the other pieces. This sum is then used to guide the search process.

3 Heuristic Calculation

The Manhattan distance for each cubelet is calculated as:

$$h = \sum_{i=1}^{n} (\text{moves required to place cubelet } i \text{ correctly})$$

where n is the total number of cubelets on the Rubik's Cube.

E. A* Search Algorithm

The **A* search algorithm** is one of the most efficient state space search algorithms for solving the Rubik's Cube, combining the advantages of both **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. It uses both the cost of the current path and a heuristic to guide the search.

1 A Overview*

A* is a **best-first search** algorithm that selects the next state to explore based on the lowest estimated total cost. This total cost, denoted as $f(n)$, is the sum of two components:

1. $g(n)$: The cost to reach the current state from the start state.
2. $h(n)$: The heuristic estimate of the cost to reach the goal from the current state.

The total cost function can be written as:

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

where $g(n)$ represents the actual cost incurred so far, and $h(n)$ is the heuristic estimate from the current state n to the goal state.

2 Components of A Search Algorithm*

3 Priority Queue

A* uses a priority queue (usually a min-heap) to store and manage states. The priority is based on the total cost $f(n)$, meaning states that seem closer to the solution (i.e., with a lower total cost) are explored first. The priority queue ensures that the most promising states, based on their heuristic value and the path taken so far, are expanded first.

4 Cost Function

The total cost $f(n)$ is determined by the actual path cost $g(n)$, combined with the heuristic estimate $h(n)$.

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

- $g(n)$ is the number of moves made from the start state to the current state.
- $h(n)$ is the estimated number of moves remaining to reach the solved state, based on the heuristic function (e.g., Manhattan distance).

5 Path Expansion

When a state is selected from the priority queue, A* generates all possible next states by applying legal cube moves (e.g., U, D, F, B, L, R). For each next state, it calculates:

- $g(n)$, which increases by 1 for each move.
- $h(n)$, which is the heuristic estimate for that new state.

Each new state is then added to the priority queue with its updated total cost $f(n)$.

6 Handling Duplicates

Since the Rubik's Cube can revisit the same state through different move sequences, A* checks if a state has already been visited with a lower cost before processing it again. If the new path is shorter, the algorithm updates the cost and the path.

7 Goal State Check

A* continues expanding states from the priority queue until the **goal state** (the solved Rubik's Cube) is reached. At that point, the algorithm terminates and returns the sequence of moves that led to the solution.

F. Efficiency and Optimality

A* is both **complete** (guaranteed to find a solution if one exists) and **optimal** (guaranteed to find the least costly solution), provided the heuristic function is **admissible**. In the case of the Rubik's Cube, the Manhattan distance is an admissible heuristic because it never overestimates the true cost to reach the goal.

1 Termination

The algorithm terminates when the goal state is dequeued from the priority queue, indicating that the shortest possible path to the solution has been found. The sequence of moves required to solve the cube is then returned.

G. Comparison with BFS

While **BFS** explores the entire state space without regard to the cost, A* focuses on states that seem closer to the goal, reducing unnecessary exploration. This allows A* to solve the Rubik's Cube more efficiently than BFS, especially when combined with a good heuristic like Manhattan distance.

VI. Conclusion

In this project, we demonstrated the application of state space search in solving the Rubik's Cube efficiently through a step-by-step approach. By dividing the solution process into three major steps—solving the first layer, the second layer, and the third layer—we were able to methodically reduce the complexity of the problem. Each layer of the cube was tackled using well-defined action sets and corresponding state spaces, which allowed for precise manipulation of cube pieces. State space exploration techniques, such as Breadth-First Search (BFS), Depth-First Search (DFS), and heuristic-based algorithms like A*, helped optimize the search process by identifying the shortest paths and reducing unnecessary moves. Additionally, the use of heuristic functions like Hamming and Manhattan distances further refined the cube-solving process, making it both efficient and scalable. This structured approach highlights the power of state space search in not only solving complex puzzles but also providing a framework for AI optimization in other strategic games.

References

[1] A comparative study of the A* heuristic search algorithm used to solve efficiently a puzzle game [CrossRef]

[2] Optimization of the game improvement and data analysis model for the early childhood education major via deep learning. [Link]

[3] The State Space of Artificial Intelligence [Link]

[4] Implementation of Rubik's Cube Algorithm[Link]