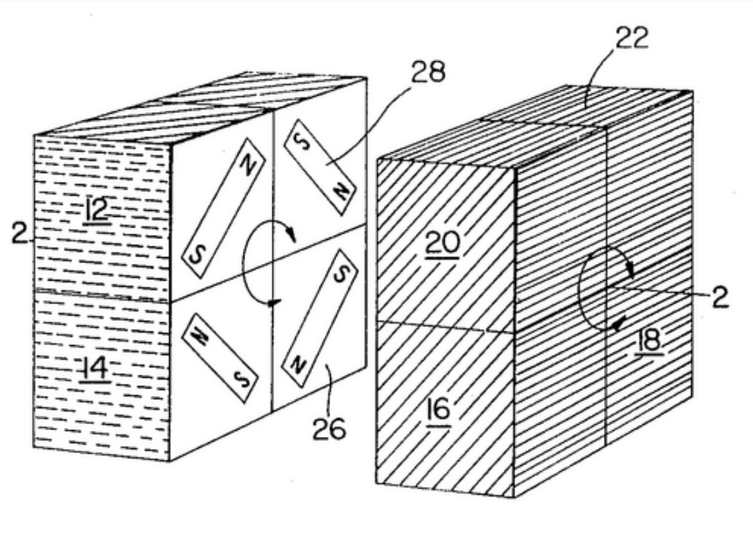
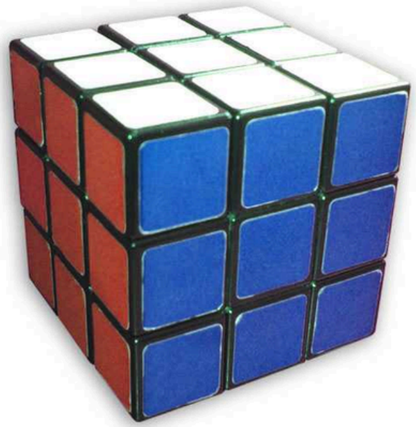
RUBIK'S CUBE SOLVER

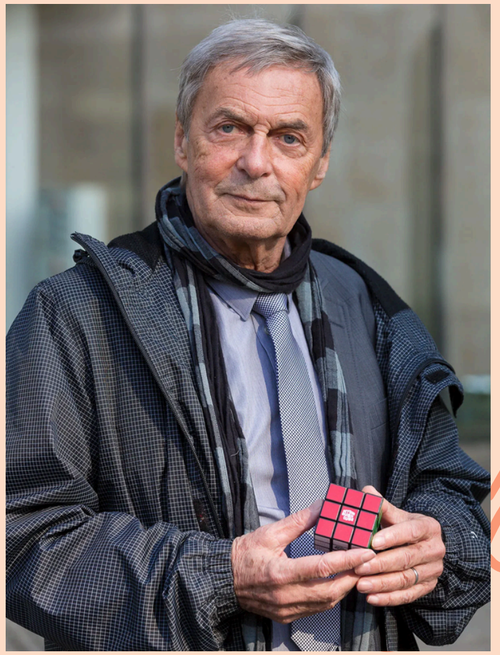
Adriatico, Chiang,
Chiu, Jakosalem, Sia

HISTORY OF RUBIK'S CUBE

- invented on May 1974 by Erno Rubik
- used to teach 3-dimensional concepts and spatial relationships
- Ideal Toy Company shared widespread international interest





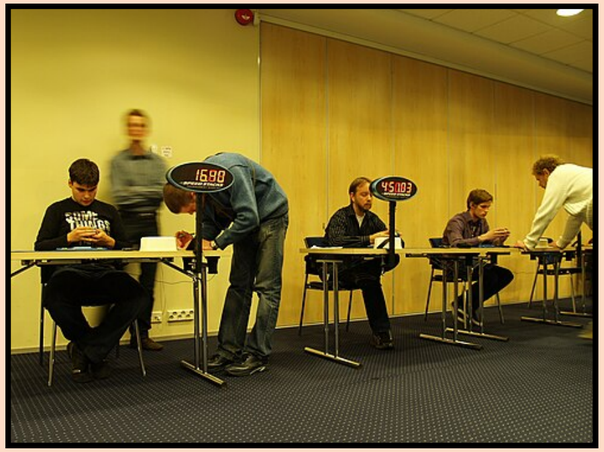


$$8! \times 3^7 \times \frac{12!}{2} \times 2^{11} = 43{,}252{,}003{,}274{,}489{,}856{,}000$$

SPEEDCUBING

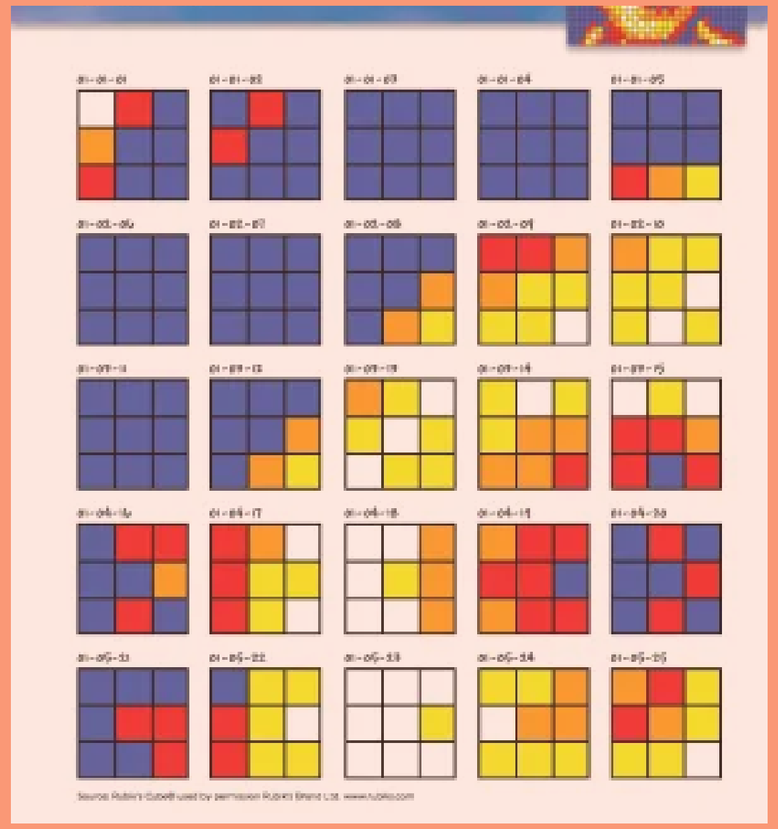
- **Speedcubing** is a competitive mind sport centered around solving a cubed puzzle using predefined algorithms with eidetic prediction

- The Rubik's Cube is not just a toy, but it is a highly-acclaimed and lucrative business model across international competitors



A speedcubing competition

WHY THIS PROBLEM?



Hardcoded Algorithms / Human-like Solvers

CFOP, Roux, ZZ - use predefined patterns and sequences, not graph search

Other Existing Algorithms

Kociemba's algorithm - extremely optimized and fast but complex, relies on prunning tables and cube symmetries

Teaching Tool using Graph-based learning

Helps students visualize abstract concepts using nodes, edges, heuristics, and optimal paths

TYPES OF ANALYSIS

1. Shortest Path
2. State Graph Traversal
3. Search Space Complexity
4. Edge Weights
5. Connectivity

A* ALGORITHM

Graph-Theoretic Model (state-space graph)

Graph Concept	Real-World Phenomenon
Vertices (Nodes)	Unique states of the Rubik’s Cube
Edges (Arcs)	Legal Moves (e.g. U, R, F, D’) transforming a state to another
Start Node	The scrambled cube state
Goal Node	Solved Cube State
Path	Sequence of Moves leading to a solution
Heuristic	Number of tiles that don’t match the center face (used in A* Search)

A* ALGORITHM

Graph-Theoretic Model (state-space graph)

Graph Theory Concept	How It's Used
State-space graph	Each cube state is a node; moves are edges
Weighted graph	Moves have assigned costs
Directed graph	Each move has a direction (U ≠ U')
Shortest path problem	Solve cube in fewest-cost steps
Heuristic search (A*)	Guide the search intelligently

Libraries with Constants and Move Mapping

- import copy
- import networkx as nx
- import matplotlib.pyplot as plt
- import matplotlib.animation as animation
- import random
- import heapq

```
import copy
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random
import heapq
```

```
# Define Rubik's Cube faces and colors
colors = ['W', 'Y', 'G', 'B', 'O', 'R']  # White, Yellow, Green, Blue, Orange, Red
faces = ['U', 'D', 'F', 'B', 'L', 'R'] # Up, Down, Front, Back, Left, Right


# Mapping each face rotation to the adjacent faces and corresponding edges that are affected
move_map = {
    'U': [('B', 0), ('R', 0), ('F', 0), ('L', 0)],
    'D': [('F', 2), ('R', 2), ('B', 2), ('L', 2)],
    'F': [('U', 2), ('R', 'col0'), ('D', 0), ('L', 'col2')],
    'B': [('U', 0), ('L', 'col0'), ('D', 2), ('R', 'col2')],
    'L': [('U', 'col0'), ('F', 'col0'), ('D', 'col0'), ('B', 'col2')],
    'R': [('U', 'col2'), ('B', 'col0'), ('D', 'col2'), ('F', 'col2')]
}
```

Cube Creation & Rotation helpers

```python
# Creates a solved Rubik's cube with each face having the same color
def create_solved_cube():
    return {face: [[color]*3 for _ in range(3)] for face, color in zip(faces, colors)}

# Rotates a 3x3 face 90 degrees clockwise
def rotate_face_cw(face):
    return [list(row) for row in zip(*face[::-1])]

# Rotates a 3x3 face 90 degrees counter-clockwise
def rotate_face_ccw(face):
    return [list(row) for row in zip(*face)][::-1]
```

Row and Column Getters and Setters

```python
# Returns a specific row from a face
def get_row(face, idx):
    return face[idx][:]

# Sets a specific row on a face
def set_row(face, idx, row):
    face[idx] = row[:]

# Returns a specific column from a face
def get_col(face, idx):
    return [row[idx] for row in face]

# Sets a specific column on a face
def set_col(face, idx, col):
    for i in range(3):
        face[i][idx] = col[i]
```

Move Application

```python
# Applies a single move to the cube (e.g., 'U', 'U'', 'R', etc.)
def apply_move(cube, move):
    face = move[0]
    direction = 1 if len(move) == 1 else -1  # 1 = clockwise, -1 = counter-clockwise

    # Rotate the face itself
    cube[face] = rotate_face_cw(cube[face]) if direction == 1 else rotate_face_ccw(cube[face])

    # Determine the affected edges
    adj = move_map[face]
    if direction == -1:
        adj = adj[::-1]

    # Extract the edges from adjacent faces
    edges = []
    for side, idx in adj:
        if isinstance(idx, int):  # It's a row
            edges.append(get_row(cube[side], idx))
        else:  # It's a column
            col_idx = int(idx[-1])
            edges.append(get_col(cube[side], col_idx))

    # Rotate the edges (shifted by one)
    edges = [edges[-1]] + edges[:-1]

    # Place the rotated edges back
    for (side, idx), edge in zip(adj, edges):
        if isinstance(idx, int):
            set_row(cube[side], idx, edge)
        else:
            col_idx = int(idx[-1])
            set_col(cube[side], col_idx, edge)
```

Scramble Generator

```python
# Applies a series of random moves to scramble the cube
def random_scramble(cube, moves=10):
    move_list = ['U', "U'", 'D', "D'", 'F', "F'", 'B', "B'", 'L', "L'", 'R', "R'"]
    scramble_seq = random.choices(move_list, k=moves)
    states = [copy.deepcopy(cube)]
    for move in scramble_seq:
        new_state = copy.deepcopy(states[-1])
        apply_move(new_state, move)
        states.append(new_state)
    return states, scramble_seq
```

State Visualizer

```python
# Draws the cube in a 2D layout using matplotlib
def visualize_cube_state(cube, title='Cube State', ax=None):
    # Layout positions for each face on a 2D grid
    face_coords = {
        'U': (3, 6), 'L': (0, 3), 'F': (3, 3),
        'R': (6, 3), 'B': (9, 3), 'D': (3, 0)
    }
    color_lookup = {
        'W': 'lightgray', # Used gray instead of white to better identify
        'Y': 'yellow',
        'G': 'green',
        'B': 'blue',
        'O': 'orange',
        'R': 'red'
    }
    if ax is None:
        fig, ax = plt.subplots()
    ax.clear()

    # Draw each square of the cube face
    for face, (x_offset, y_offset) in face_coords.items():
        face_grid = cube[face]
        for i in range(3):
            for j in range(3):
                color = color_lookup[face_grid[i][j]]
                ax.add_patch(plt.Rectangle(
                    (x_offset + j, y_offset + 2 - i), 1, 1, facecolor=color))
    ax.set_xlim(0, 12)
    ax.set_ylim(0, 9)
    ax.set_aspect('equal')
    ax.set_title(title)
    ax.axis('off')
```

Animation

```python
# Animates a list of cube states as a solution is applied
def animate_solution(states, title='Solving Cube'):
    fig, ax = plt.subplots()

    def update(frame):
        visualize_cube_state(states[frame], title=f'{title}: Step {frame}', ax=ax)

    ani = animation.FuncAnimation(
        fig, update, frames=len(states), repeat=False, interval=500)
    plt.show()
```

Utility Functions

```python
# Checks if the cube is in a solved state
def is_solved(cube):
    for face in cube:
        center = cube[face][1][1]
        for row in cube[face]:
            for color in row:
                if color != center:
                    return False
    return True

# Heuristic function for A* search – counts how many stickers are not the same as the center color of their face
def heuristic(cube):
    score = 0
    for face in cube:
        center = cube[face][1][1]
        for row in cube[face]:
            for color in row:
                if color != center:
                    score += 1
    return score

# Applies a sequence of moves to the cube and returns all intermediate states
def apply_sequence(cube, sequence):
    states = [copy.deepcopy(cube)]
    for move in sequence:
        apply_move(cube, move)
        states.append(copy.deepcopy(cube))
    return states
```

A* Search Algorithm

```python
# Solves the scrambled cube using A* search
def a_star_solver(start_cube):
    move_list = ['U', "U'", 'D', "D'", 'F', "F'", 'B', "B'", 'L', "L'", 'R', "R'"]
    visited = set()
    queue = []
    counter = 0

    # Push initial state into priority queue (min-heap): (estimated_total_cost, current_cost, counter, cube, path)
    heapq.heappush(queue, (0, 0, counter, start_cube, []))

    while queue:
        _, cost, _, current, path = heapq.heappop(queue)
        cube_id = str(current)  # Serialize cube for visited set

        if cube_id in visited:
            continue
        visited.add(cube_id)

        if is_solved(current):
            return path  # Return the solution path

        for move in move_list:
            new_cube = copy.deepcopy(current)
            apply_move(new_cube, move)
            if str(new_cube) not in visited:
                new_path = path + [move]
                est_cost = cost + 1 + heuristic(new_cube)
                counter += 1
                heapq.heappush(queue, (est_cost, cost + 1, counter, new_cube, new_path))
    return []
```

Main Execution

```
# Initialize the cube / start with the solved state
cube = create_solved_cube()

# Scramble the cube randomly
states, scramble_seq = random_scramble(cube, moves=5)
scrambled = copy.deepcopy(states[-1])
print("Scramble sequence:", scramble_seq)

# Visualize both states
visualize_cube_state(states[0], title='Solved State') # Shows the initial solved cube
visualize_cube_state(scrambled, title='Scrambled State') # Shows the scrambled state

# Solve the scrambled cube using A* algorithm
solution_seq = a_star_solver(copy.deepcopy(scrambled))
print("Solution sequence:", solution_seq)

# Apply the solution sequence to the scrambled cube to generate the solving steps
solve_states = apply_sequence(scrambled, solution_seq)

# Animate the solving process, step-by-step
animate_solution(solve_states, title='Solving with A* Search')
```

DEFINING A*

f(n) = g(n) + h(n)
estimated total cost

- It avoids unnecessary moves
- It chooses moves that get closer to a solved state and doesn't take too many step

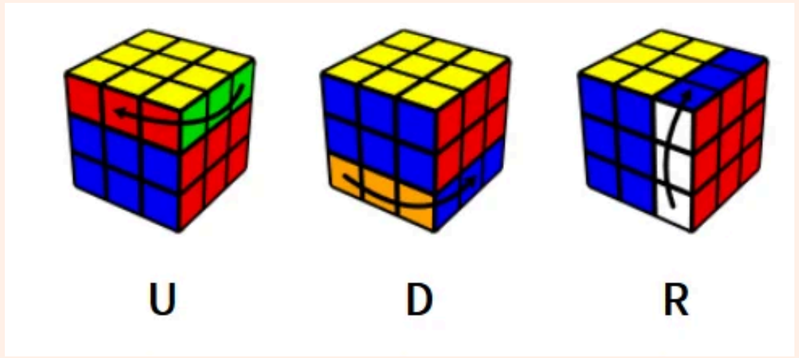
g(n) = accumulated path cost

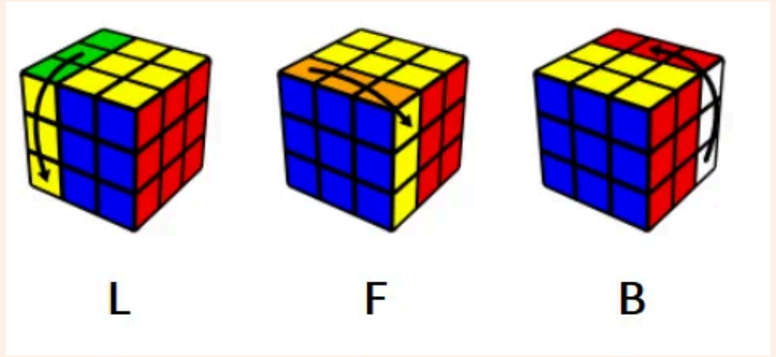
number of moves taken so far (actual cost)

h(n) = heuristic value

number of mismatched stickers not matching the center colors (heuristic)

CUBE NOTATIONS





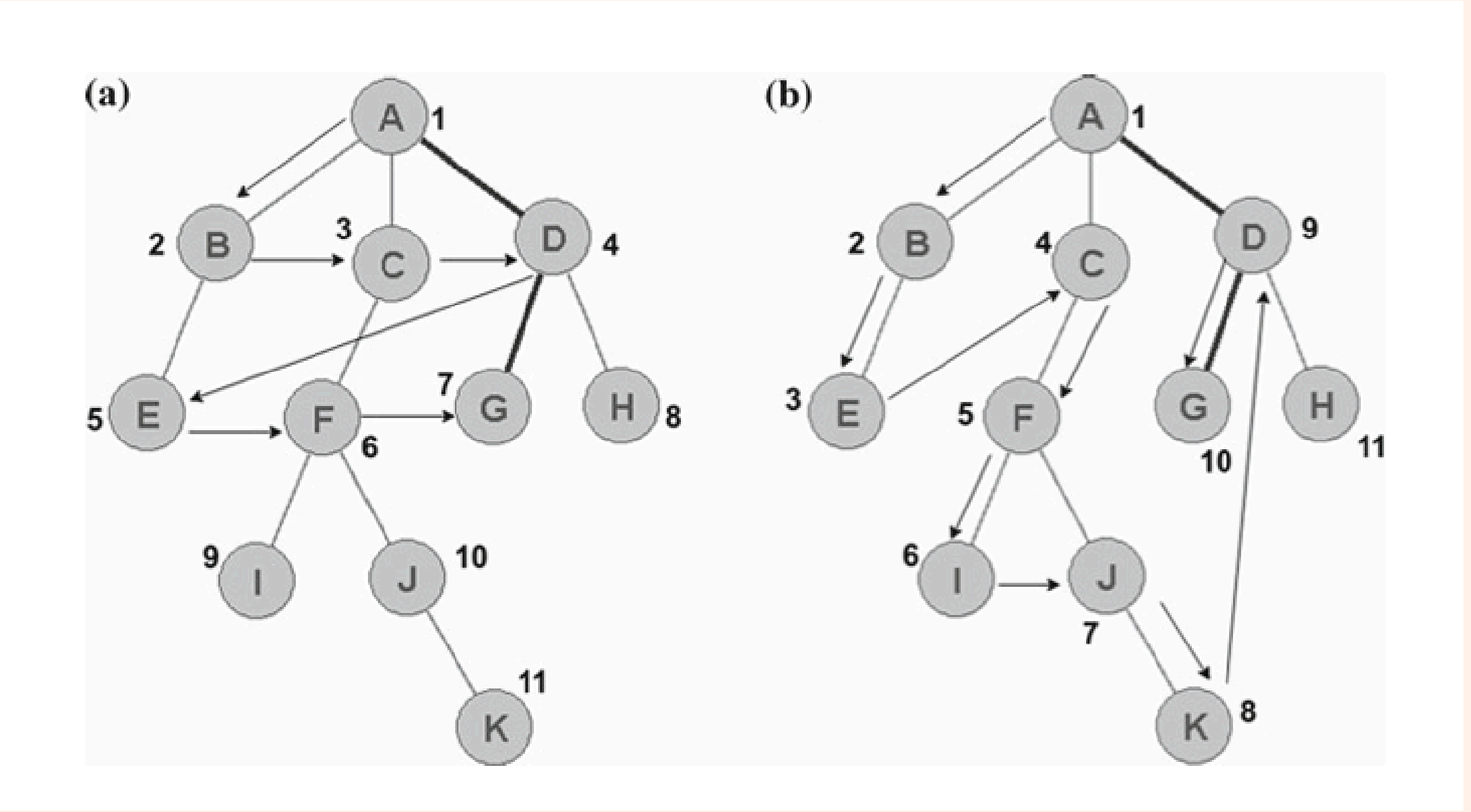
Take note:

by default, it goes through a clockwise notation a letter denoted with ‘ means counterclockwise ex: U’

PAST STUDIES ON RUBIK'S CUBE SOLVING ALGORITHMS

(PDF) Solving Rubik's Cube Using Graph Theory: ICCI-2017

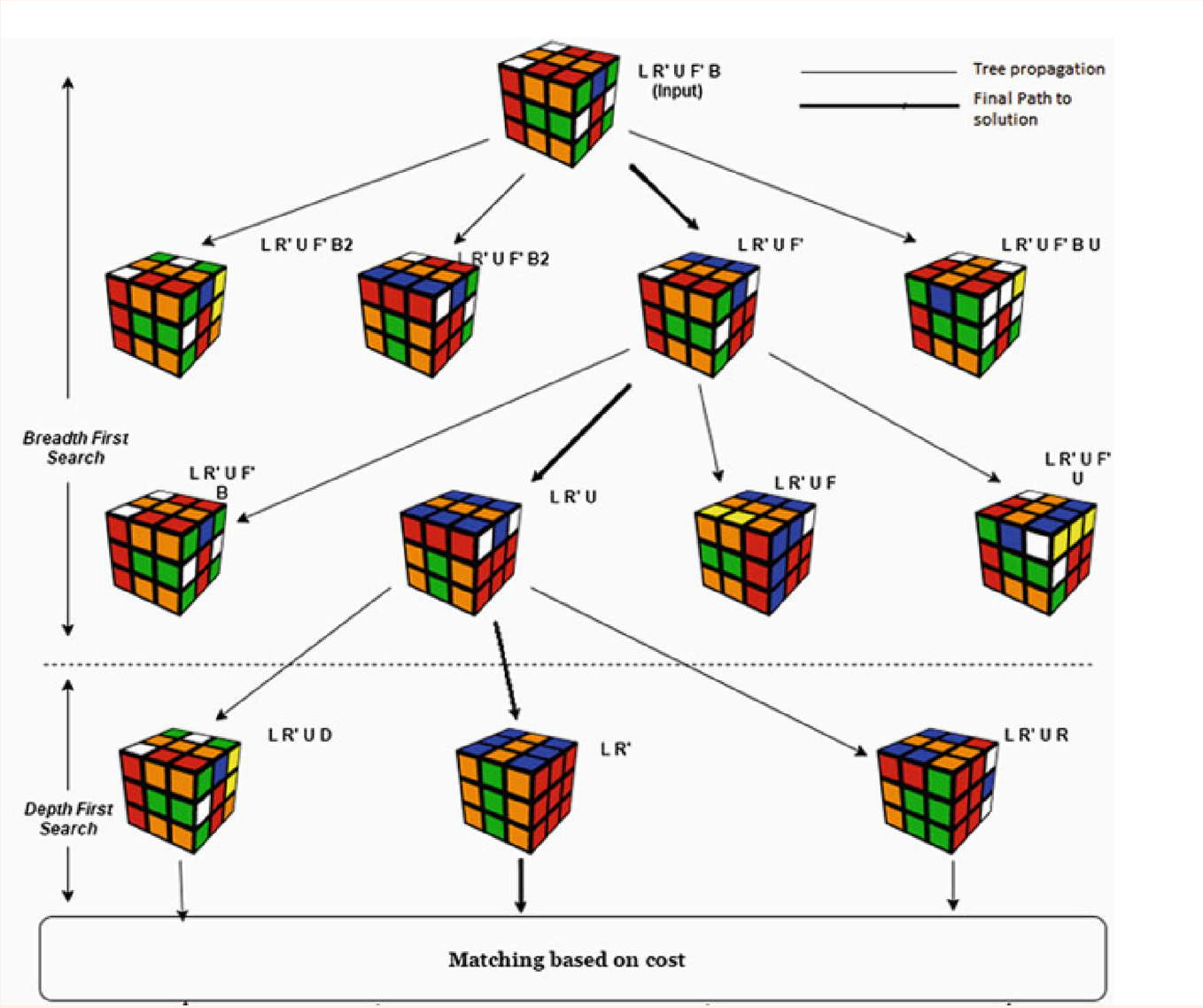
a Breadth-first search.
b Depth-first search



PAST STUDIES ON RUBIK'S CUBE SOLVING ALGORITHMS

(PDF) Solving Rubik's Cube Using Graph Theory: ICCI-2017

Breadth-first search,– Parallel depth-limited search, and– Matching phase



PAST STUDIES ON RUBIK'S CUBE SOLVING ALGORITHMS

(PDF) Solving Rubik's Cube Using Graph Theory: ICCI-2017

Breadth-first search,– Parallel depth-limited search, and– Matching phase

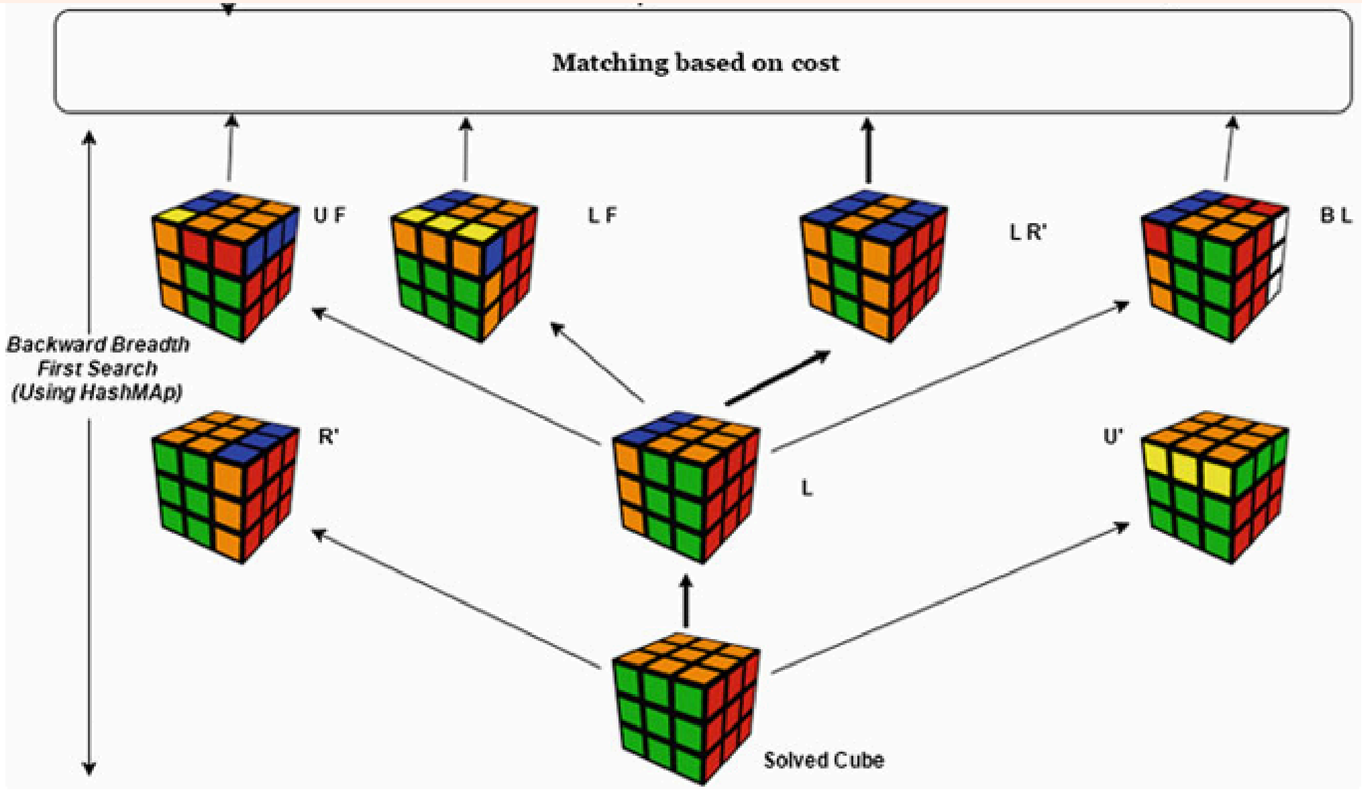
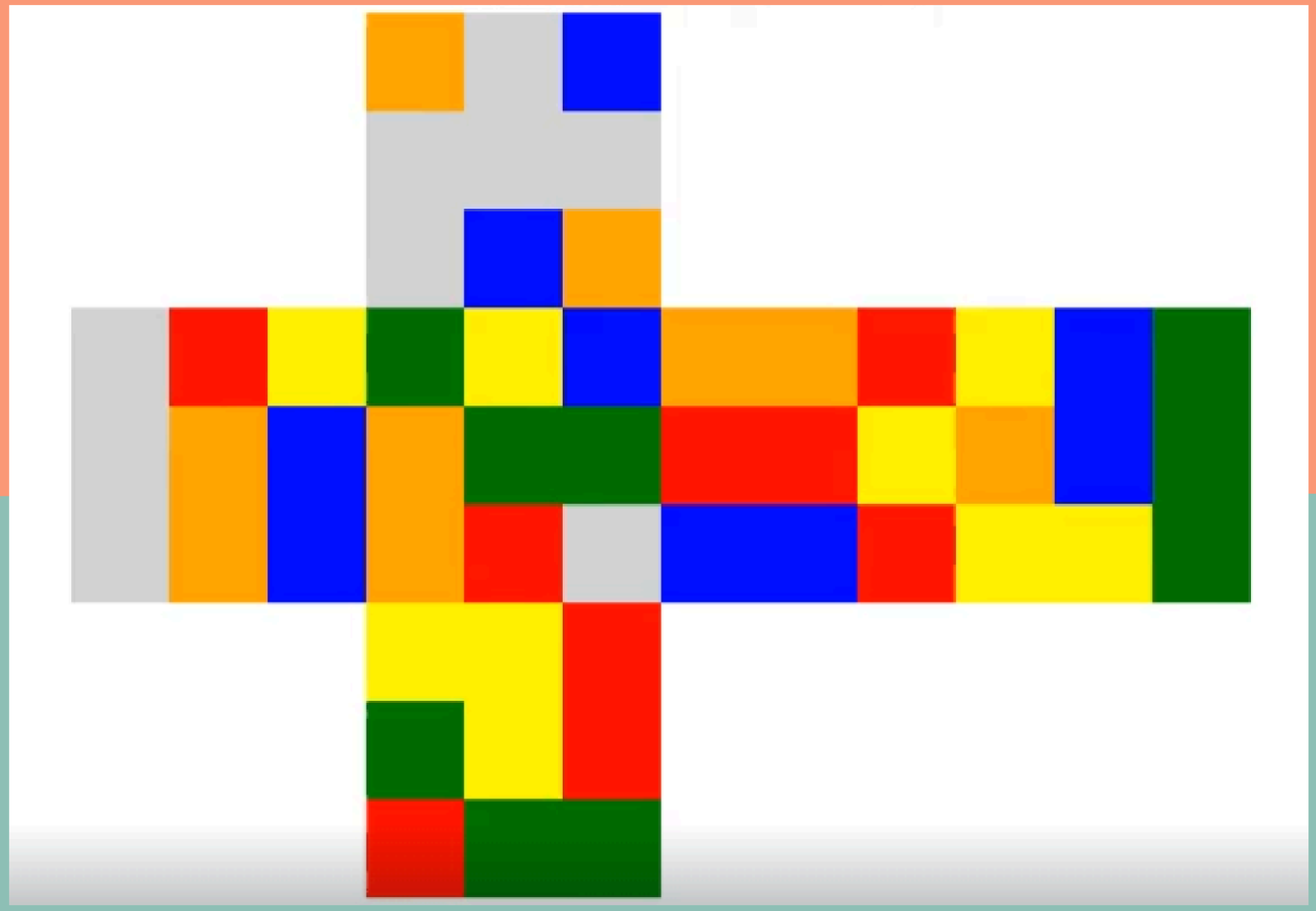


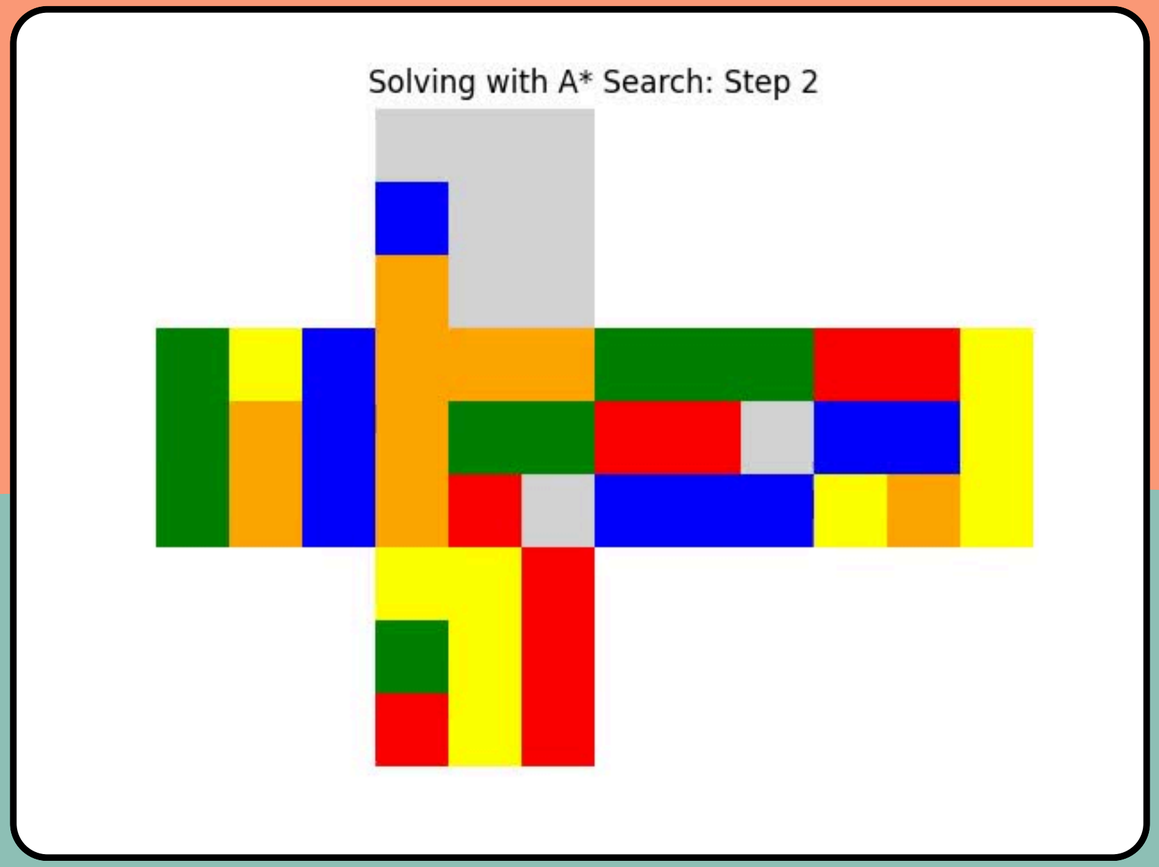
Fig. 4 Implementation flow example. Input scrambled cube: $LR'UF'B$

RUBIKS CUBE SCRAMBLED



using sequence: [“B'”, “D'”, “U'”, 'L', “U'”, “F'”, 'F', 'B', “L'”, 'L']

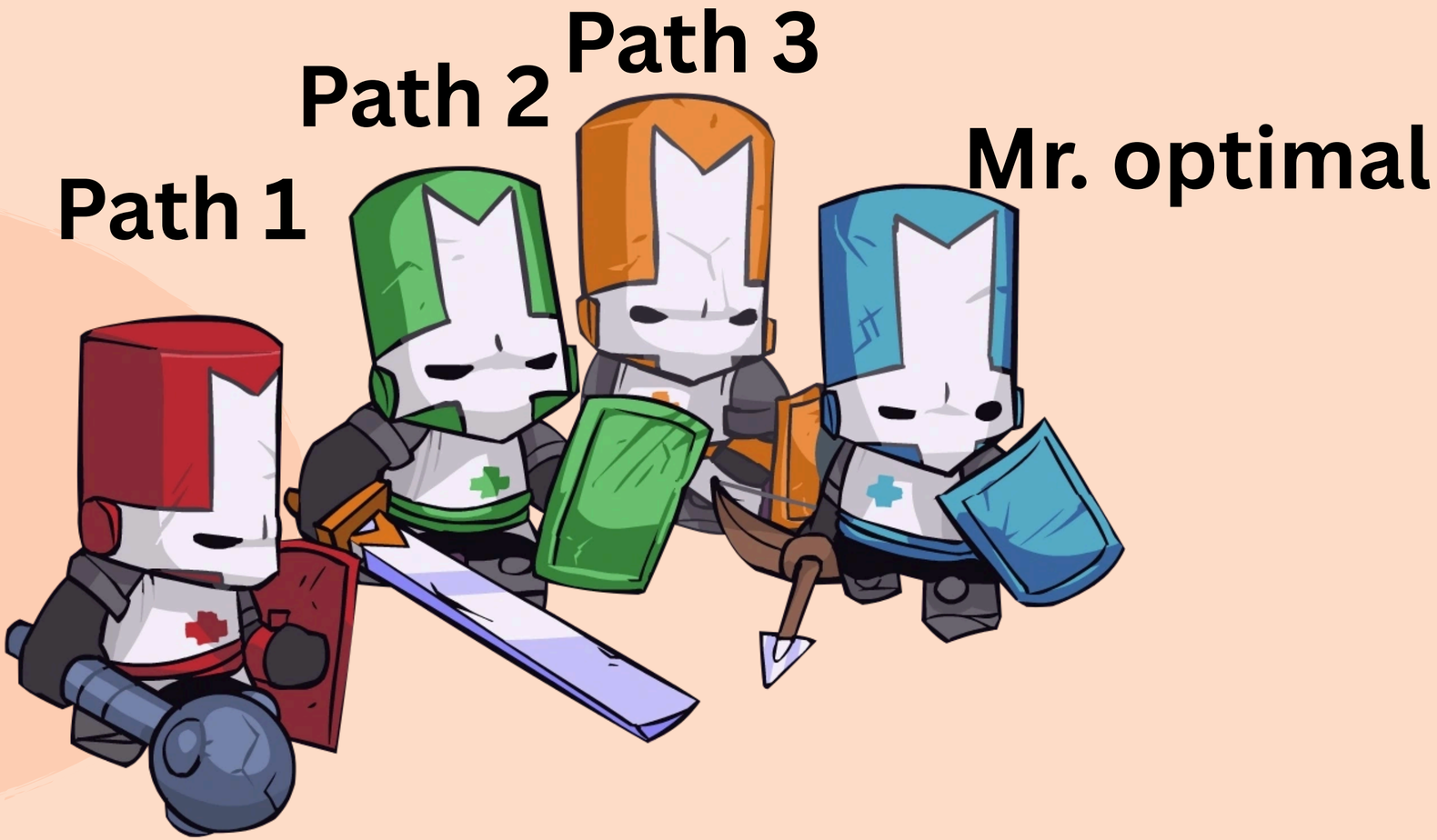
RUBIKS CUBE SOLVED



using sequence [“B'”, 'U', “L'”, 'U', 'D', 'B']

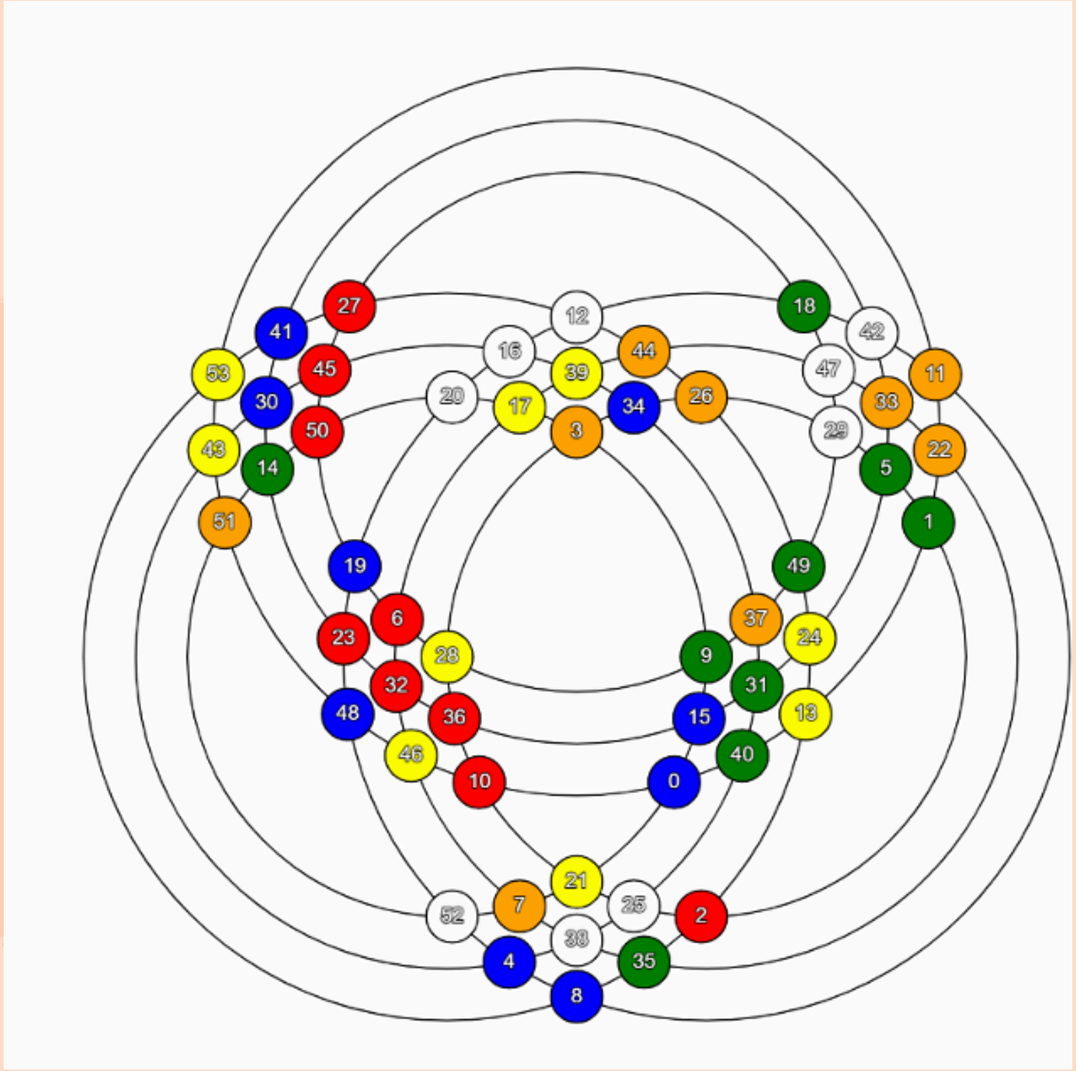
RUBIK'S CUBE GRAPH

the one with the least points wins!

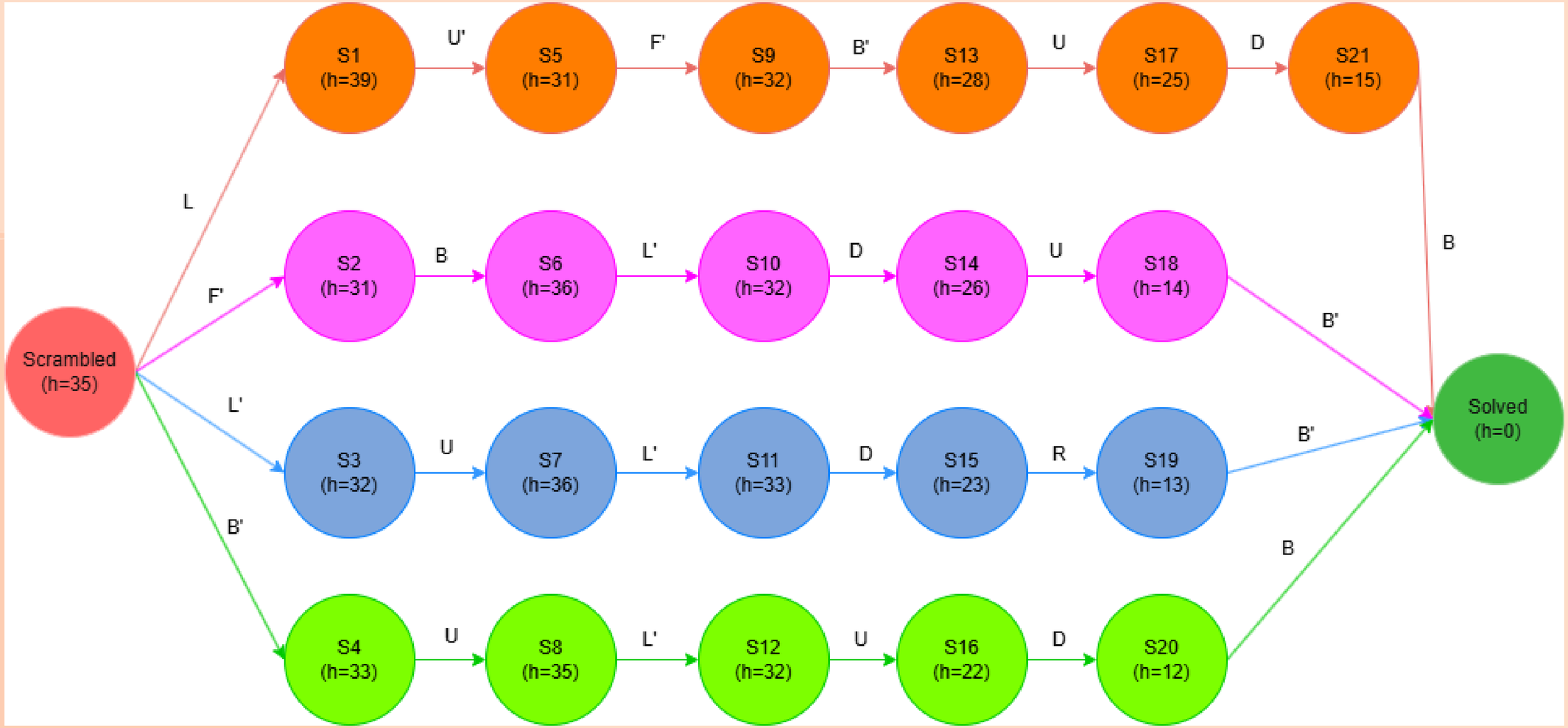


RUBIK'S CUBE GRAPH

Sample Overview



RUBIK'S CUBE GRAPH



PATH #1

Table Weights

Step	Move	h(n) (Heuristic)	g(n) (Cost)	f(n) = g(n) + h(n)
0	—	35	0	35
1	L	39	1	40
2	U’	31	2	33
3	F’	32	3	35
4	B’	28	4	32
5	U	25	5	30
6	D	15	6	21
7	B	0	7	7 ✅ Goal reached

Total Points

233

PATH #2

1 233

Table Weights

Step	Move	h(n) (Heuristic)	g(n) (Cost)	f(n) = g(n) + h(n)
0	—	35	0	35
1	F’	31	1	32
2	B	36	2	38
3	L'	32	3	35
4	D	26	4	30
5	U	14	5	19
6	B’	0	6	6 ✅ Goal reached

Total Points

195

PATH #3

Table Weights

1	233
2	195

Step	Move	h(n) (Heuristic)	g(n) (Cost)	f(n) = g(n) + h(n)
0	—	35	0	35
1	L’	32	1	33
2	U	36	2	38
3	L'	33	3	36
4	D	23	4	27
5	R	13	5	18
6	B’	0	6	6 ✅ Goal reached

Total Points

193

PATH 4 (OPT.)

Table Weights

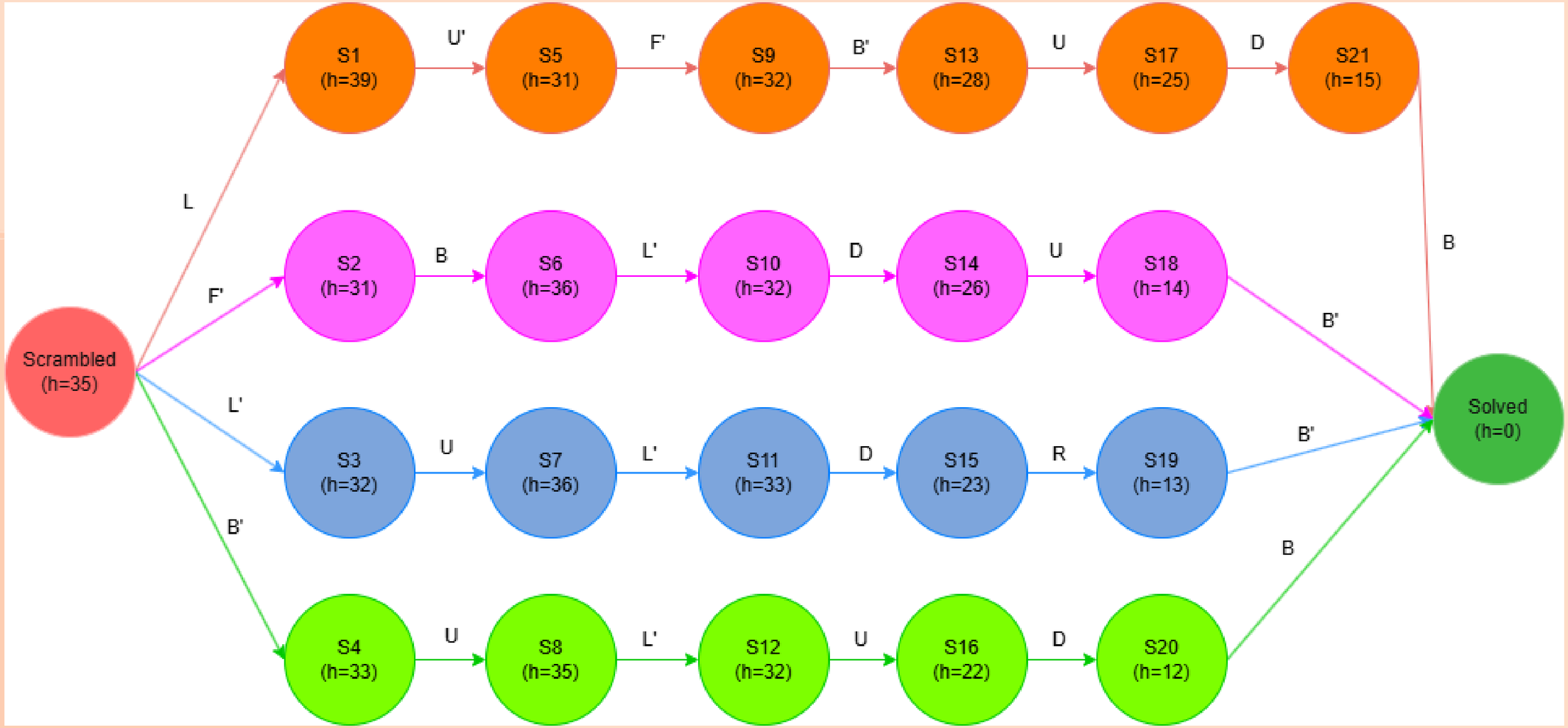
1	233
2	195
3	193

Step	Move	h(n) (Heuristic)	g(n) (Cost)	f(n) = g(n) + h(n)
0	—	35	0	35
1	B'	33	1	34
2	U	35	2	37
3	L'	32	3	35
4	U	22	4	26
5	D	12	5	17
6	B	0	6	6 ✅ Goal reached

Total Points

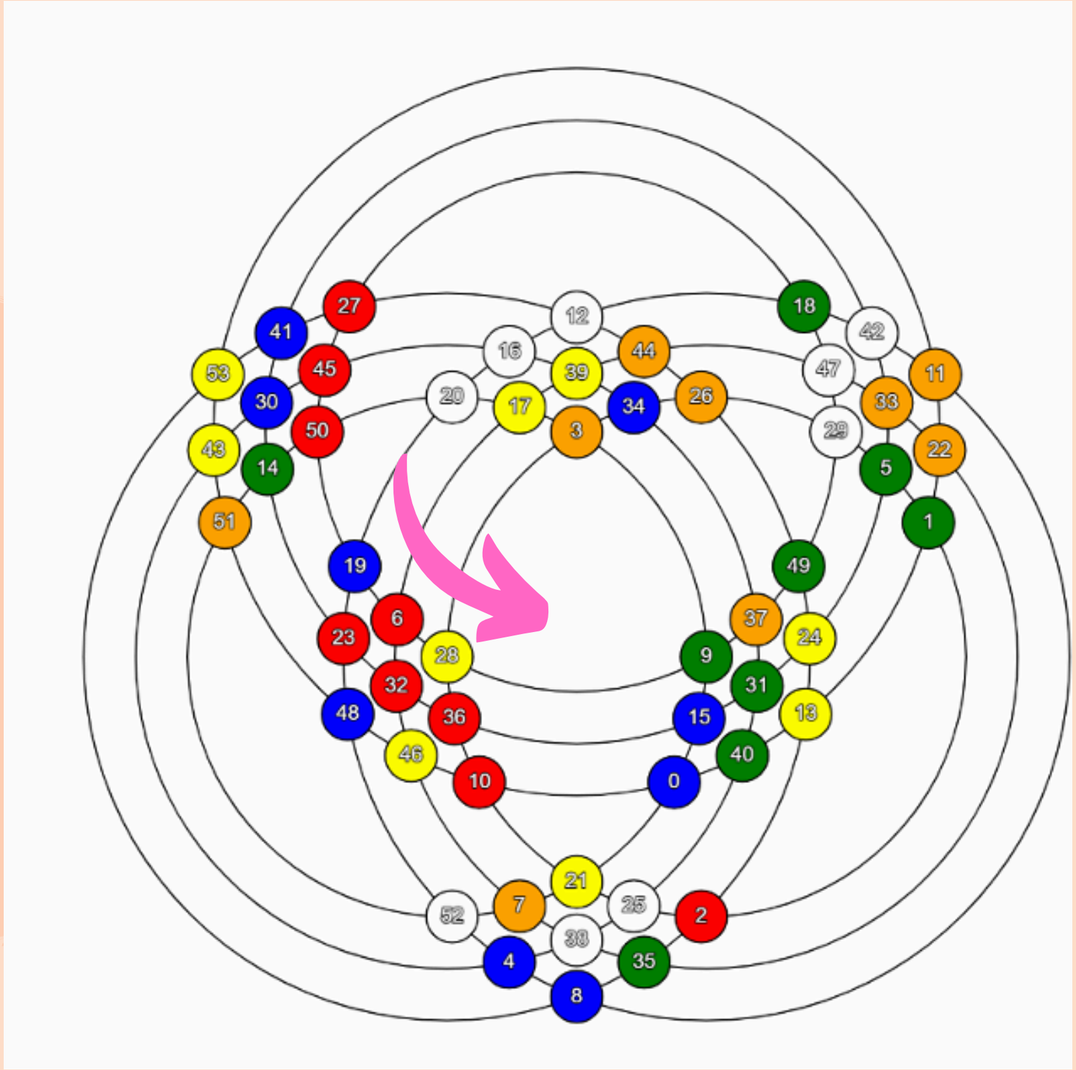
190

RUBIK'S CUBE GRAPH



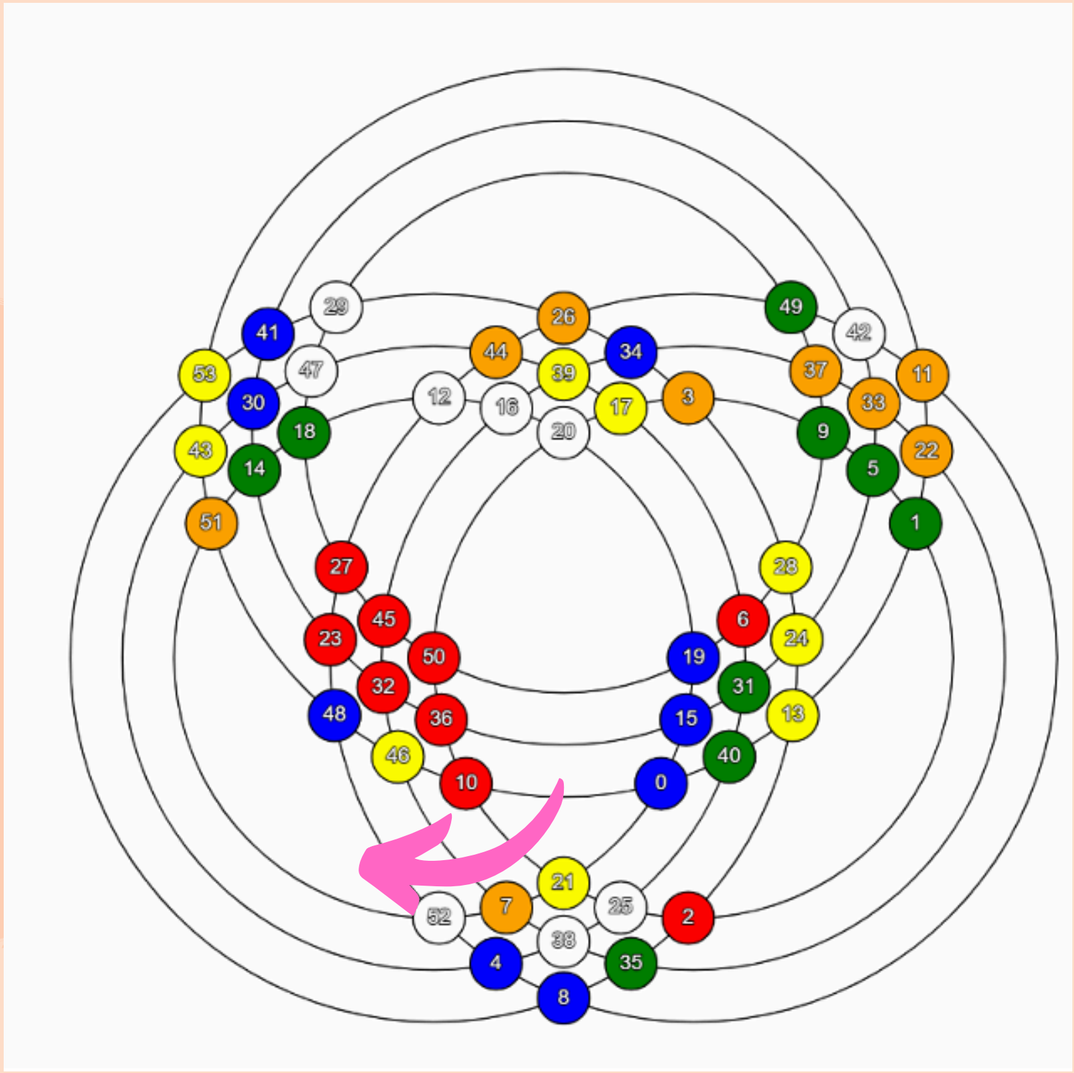
RUBIK'S CUBE GRAPH

Next MOVESET: B'



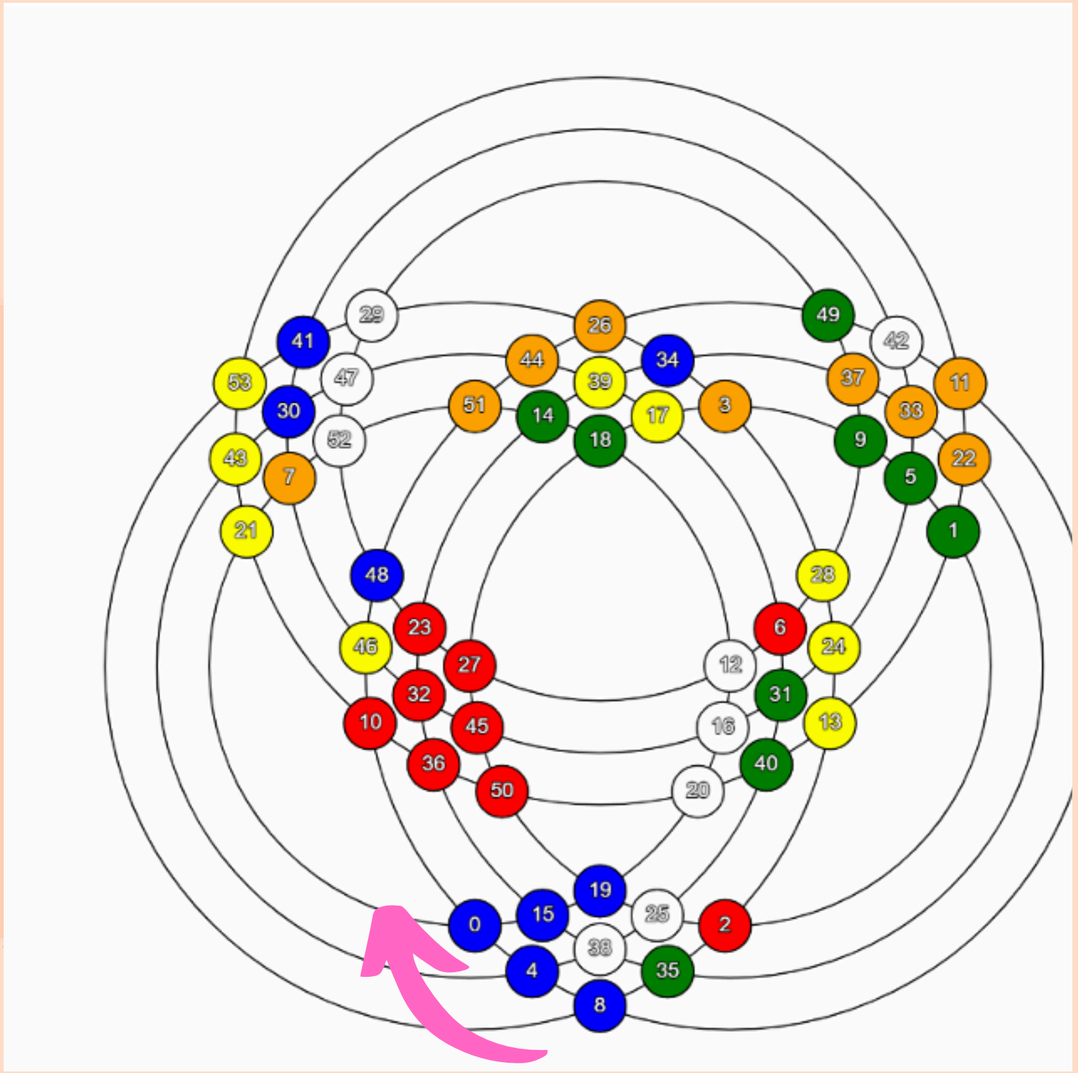
RUBIK'S CUBE GRAPH

Next MOVESET: U



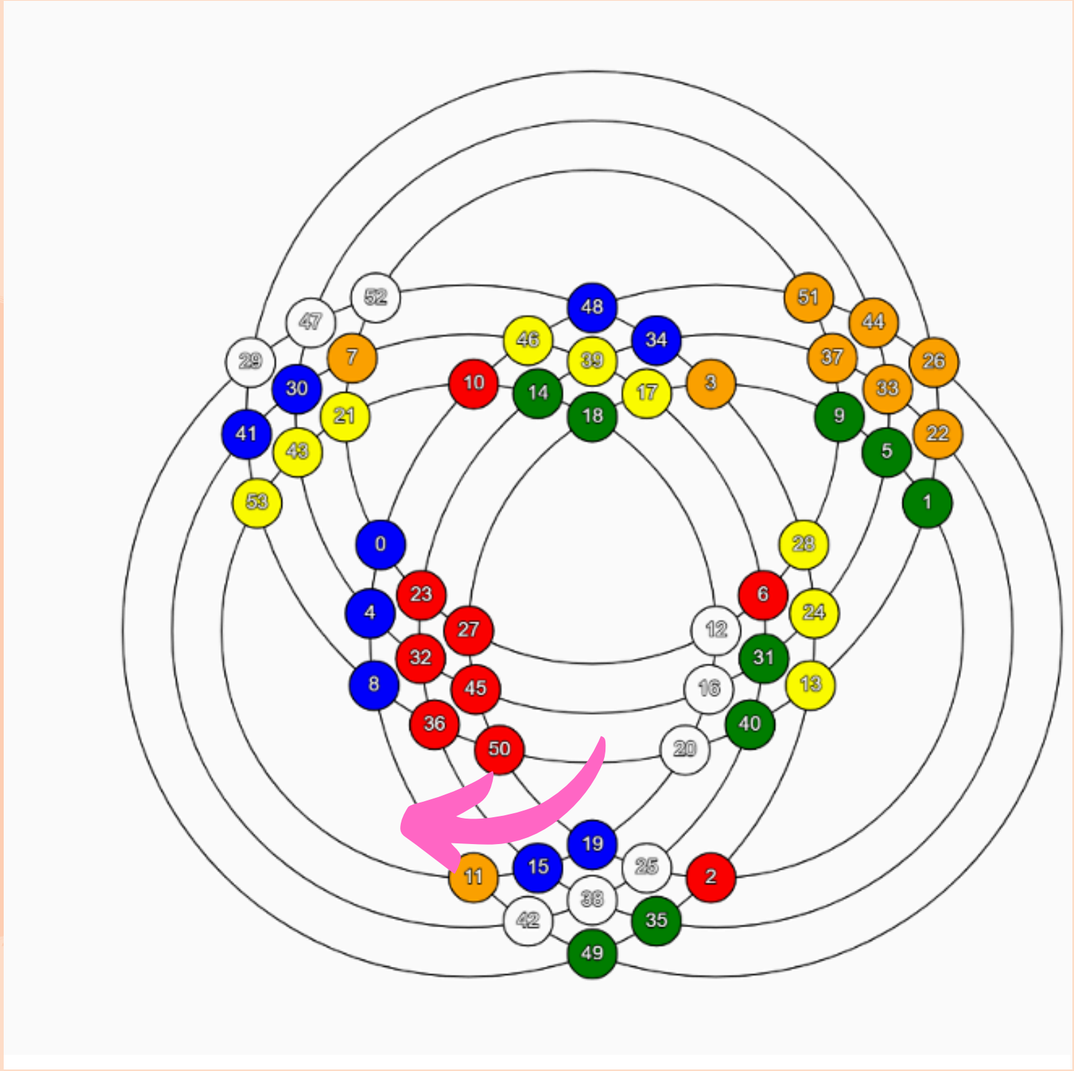
RUBIK'S CUBE GRAPH

Next MOVESET: L'



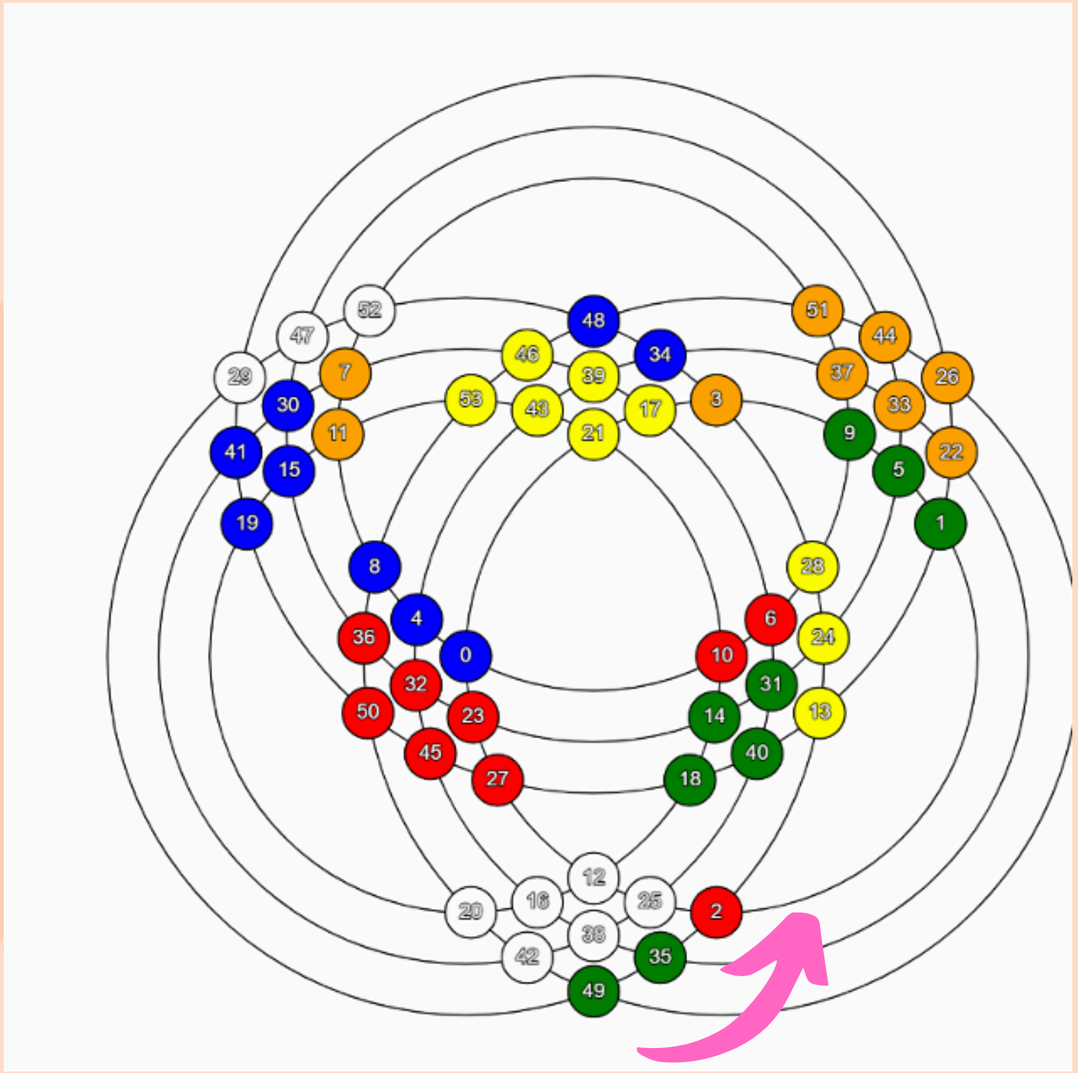
RUBIK'S CUBE GRAPH

Next MOVESET: U



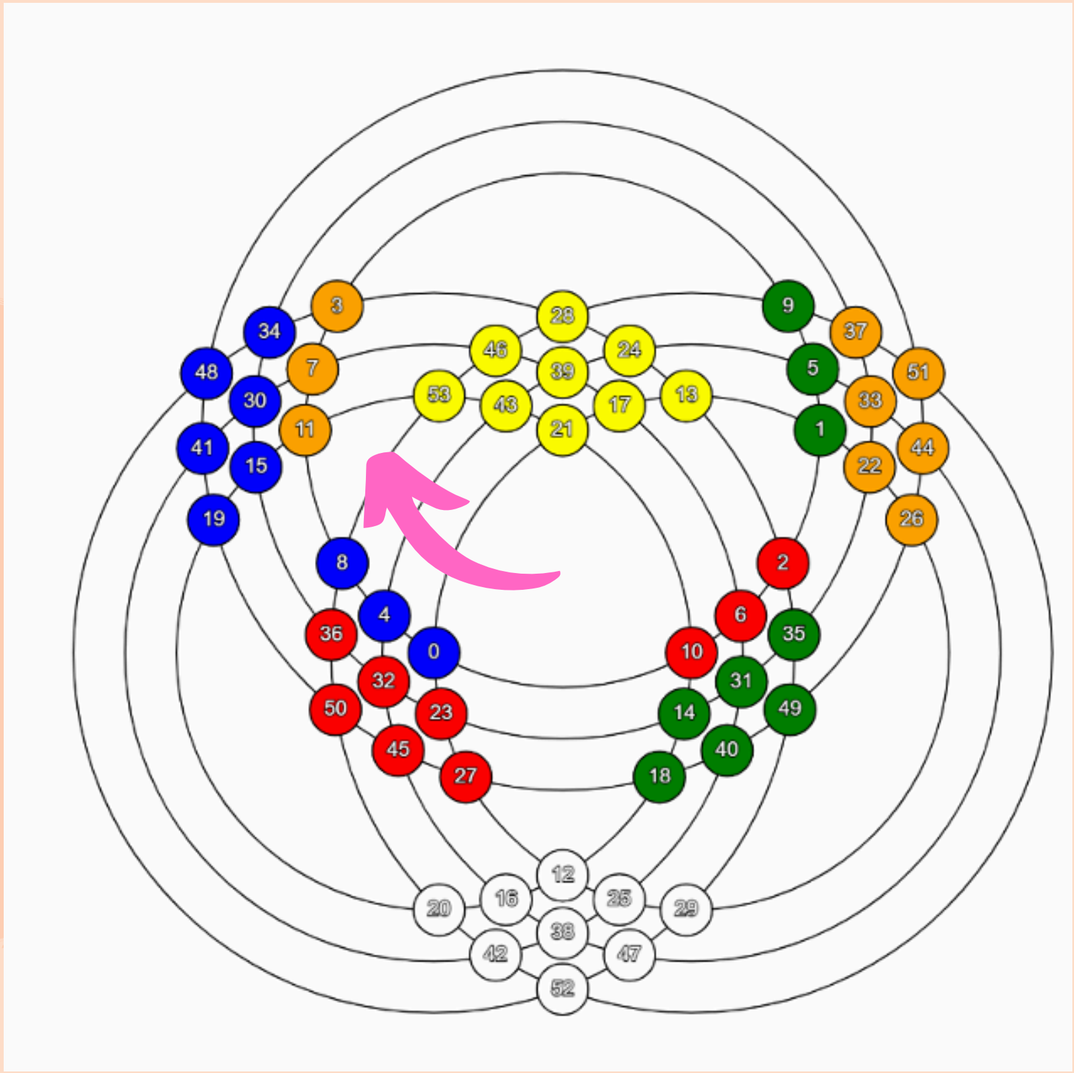
RUBIK'S CUBE GRAPH

Next MOVESET: D

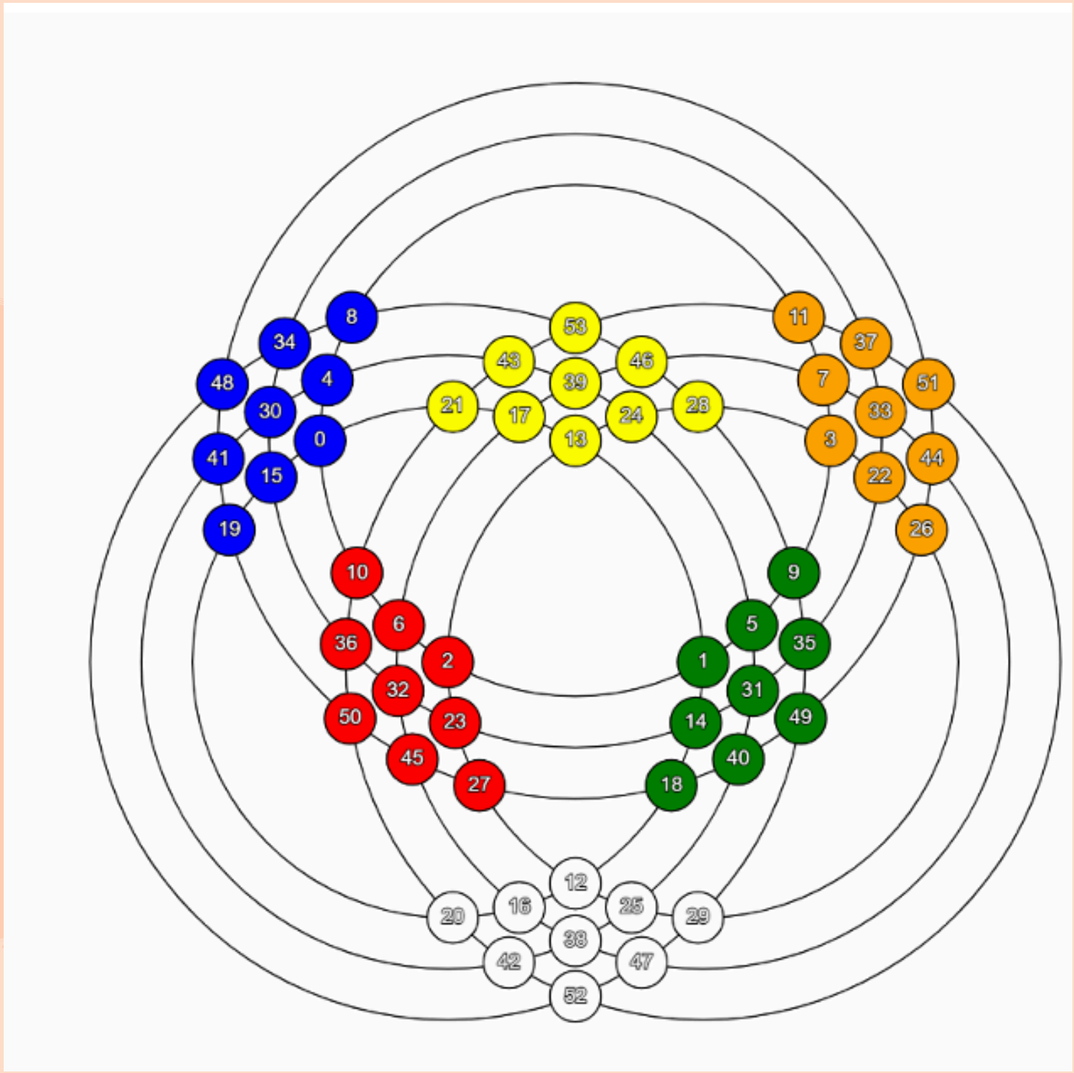


RUBIK'S CUBE GRAPH

Next MOVESET: B



RUBIK'S CUBE GRAPH



THANK YOU