Anton Florendo
301427524
CMPT 225 D100
Prof. Igor Shinkar
Fall 2025

CMPT 225 Final Project - Rubik’s Cube Solver

Introduction

This document explains my implementation of a Rubik’s Cube Solver for the Final Project in CMPT 225, Fall 2025. The solver uses the Kociemba Two-Phase Algorithm to take a scrambled cube as the input and generate a sequence of moves to solve it. The final working solution was achieved using an iterative development process that explored and refined different approaches.

The standard 3x3x3 Rubik’s Cube is very complex, with about 43 quintillion possible arrangements. If a cube needs more than a few turns, using a basic, brute-force approach will not get far. That is why the Two-Phase Algorithm is efficient, as it simplifies the problem by breaking it into two smaller steps.

Initial Approach: Corner Pattern Database

My initial implementation used a single-phase IDA* search with a corner database. The database was developed using the BFS search algorithm, storing corner configurations in a HashMap<String, Integer> mapping each configuration to its distance from the solved state. This approach used around 50 MB of memory for a database with a depth of 5.

Optimization Attempts

I tried numerous optimizations to improve my first approach:

- Apply/unapply moves instead of copying
- char[] for path instead of String concatenation
- Move Pruning (avoiding F->F’, and commutative moves like F->B)
- Long hash keys instead of String keys

Combined, these optimizations provided a good speedup. But this was insufficient for deeply scrambled cubes like scrambles 04-09, because the fundamental issue was the weak heuristic, not the implementation speed.

Transition to Kociemba

The corner-only heuristic had a maximum depth of 5 moves, meaning the search was blind for positions which required 15+ moves. What I learned was that the optimization cannot fix a weak heuristic. Kociemba’s algorithm provides heuristics with depths of 10-12 moves, providing effective pruning especially for deeply scrambled cubes.

The Two-Phase Algorithm

Theoretical Foundation
The algorithm uses the Rubik's Cube's group structure. The whole cube has 43 quintillion elements, but the subgroup G1 (cubes with all pieces oriented correctly and E-slice edges in the middle layer) has only 2.2 billion elements. Phase 1 uses all 18 possible moves to reach G1, where Phase 2 only uses moves that preserve G1 to solve the cube.

Phase 1: Reaching G1
Phase 1 transforms the cube so that:
- All corners are oriented correctly (twist = 0)
- All edges are oriented correctly (flip = 0)
- E-slice edges (FR, FL, BL, BR) are in the middle layer

All 18 moves (U, U2, U', R, R2, R', F, F2, F', D, D2, D', L, L2, L', B, B2, B') can be used. The coordinates used are twist, flip, and slice positions.

Phase 2: Solving from G1
Phase 2 uses restricted moves that preserve G1:
- U, U2, U' and D, D2, D' (Up and Down moves)
- R2, L2, F2, B2 (Half-turns on other faces)

The coordinates used are the corner permutation (20, 160 values), the UD-edge permutation (20, 160 values), the E-slice permutation (24 values), and parity (2 values).

Architecture and Data Structures

Class Overview

Class	Responsibility
Solver	Handles I/O and verification.
RubiksCube	Parses the input file and converts it to Cubie representation.
Cubie	The cubie-level representation provides coordinates and a conversion from the facelet representation.
Tables	Precomputes all move tables and pruning tables in memory at runtime.
TwoPhase	IDA* search implementation for both phases.

Cubie Representation
The Cubie class represents the cube using four arrays: cornerPerm[8] (corner piece at each position), cornerOrient[8] (twist of each corner: 0, 1, or 2), edgePerm[12] (edge piece at each position), and edgeOrient[12] (flip of edge: 0 or 1). Moves are applied using a permutation multiplication.

Facelet to Cubie Conversion
The RubiksCube class reads 54 sticker colours from the input file and converts them to the Cubie representation. For each corner position, it reads the three sticker colours, finds which one is Red or Orange (U/D colour) to determine the twist (0, 1, or 2), then matches the other two colours to identify which of the eight corner pieces it is. For edges, it is similar: it reads two sticker colours, matches them to the 12 edge pieces, and checks whether the colours are in order or reversed.

Coordinate Systems
Has O(1) table lookups:

Coordinate	**Range**	**Derivation**
Twist	0 - 2, 186	3^7 = 2 187 (7 corners, 8th determined)
Flip	0 - 2, 047	2^{11} = 2 048 (11 edges, 12th determined)
Slice Position	0 - 494	495 positions for 4 E-slice edges
Corner Permutation	0 - 20, 159	20, 160
UD Edge Permutation	0 - 20, 159	20, 160
Slice Permutation	0 - 23	4! = 24 arrangements of E-slice edges

Move Tables
Move tables, precompute the result of applying each of the 18 moves to each coordinate. This transforms the move application from O(n) permutation operations to O(1) array lookups.

Pruning Tables and Heuristic

Generation using Backward BFS
Pruning tables are built using BFS starting from the solved state and working backward.
Process:

1. Initialize the solved state with coordinate zero and depth 0.
2. For each state at depth D, apply all 18 moves and mark any unvisited states with depth D+1.

3. Repeat until all reachable states have been assigned a depth. Since we are using BFS, the first time we reach any state, it is guaranteed to be the shortest path.

Combined Coordinates
Rather than storing separate tables for each coordinate, the implementation combines coordinates to create a stronger heuristic. For example, sliceTwistPrune indexes by (slice x 2187 + twist). A state that only has a pruning value of 0 when both the slice position and corner twist are solved.

The max() Heuristic
The heuristic returns the maximum of table lookups h = max(sliceTwistPrune[...], sliceFlipPrune[...]). If twist + slice needs five moves and flip + slice needs eight moves, we need at least eight moves total. Taking the maximum is admissible provided it yields more vigorous pruning than any single table alone.

Memory Optimization
Values are stored using Half-Byte packing (two 4-bit values per byte), since pruning depths rarely exceed 15. Total memory usage is approximately 2 MB for all pruning tables.

Implementation Comparison

Aspect	Old Implementation	New Implementation
Algorithm	Single-phase IDA*	Two-phase Kociemba Algorithm
Data Structure	HashMap<String, Integer>	2D arrays (short[][])
Heuristic	Corner pattern only	Combined coordinate pruning
Max Heuristic Depth	~5 moves	~12 moves (Phase 1)
State Space	~10^{19} states	~10^{7} + ~10^{6} states
Lookup Complexity	O(k) hash + equals	O(1) array index
Memory Usage	~50 MB HashMap	~3.5 MB arrays

Why the Old Approach Failed
The HashMap-based approach did not succeed due to the “one aspect solved” problem. The corner pattern database only tracked corner positions and orientations, and ignored edges entirely. When corners were correctly configured, edges remained unscrambled, forcing a blind search.

Complexity Analysis

Time complexity

- **Table Generation:** O(N x 18 x D) - where N is the coordinate space, and D is the maximum depth
- **Search:** O(1) per node for coordinate updates and heuristic lookups

Space complexity

- **Move Tables:** ~1.5 MB total
- **Pruning Tables:** ~2 MB total
- **Search Stack:** O(d) - where d <= 30 - 40 on average (max solution length)

Challenges and Lessons Learned

Difficulties Faced

1. **Edge Orientation Definition:** The Kociemba edge orientation rule is highly position-dependent and not based solely on sticker colours. Getting this wrong caused incorrect coordinates.
2. **Coordinate Encoding:** The mapping between cube states and integers required careful and precise implementation of combinatorial and factorial number systems.
3. **Move Direction Bug:** An early bug was causing some of my moves to be implemented incorrectly. For example, My U move was implemented counter-clockwise instead of clockwise, which caused many errors, especially with solutions that required the use of U. This was especially evident when cubes did not match database entries.

Lessons I Learned

- Optimization cannot compensate for a weak heuristic; it requires algorithm improvements and rework to implement successfully.
- Mathematical coordinate encoding is more advantageous than string-based state representation.
- The cubie representation enables efficient coordinate extraction and movement.
- Different search algorithms like A*, IDA*, and bidirectional BFS serve various purposes and have their own advantages, depending on implementation.

References

- Edwards, S., Tian, H., Brera, R., & Rosenberg, M. (n.d.). *Rubik's Cube Solver using IDA** [Class project]. https://www.cs.columbia.edu/~sedwards/classes/2024/4995-fall/reports/rubik-presentation.pdf
- The Two-Phase-Algorithm. (n.d.). Kociemba.org. https://kociemba.org/math/twophase.htm
- Khan, M. A. (2025, September 11). *Kociemba's Two-Phase Algorithm: How It Works and Its Applications*. Medium. https://medium.com/@muhammadalikhan0003/kociembas-two-phase-algorithm-how-it-works-and-its-applications-3d8f97a3562a
- Korf, R. E. (1997). *Finding optimal solutions to Rubik's Cube using pattern databases.* https://www.cs.princeton.edu/courses/archive/fall06/cos402/papers/korfrubik.pdf
- Feldhausen, R. (2008). *Implementing a two-phase algorithm for solving a Rubik's Cube* [Honours project presentation]. Kansas State University. https://russfeld.me/projects/rubiks/presentation.pdf