

CMPT 225 Final Project: Master Implementation Plan

Optimized IDA* Solver

Architecture & Code Guide

November 26, 2025

1 Architecture Overview

Goal: Solve a Rubik's Cube in < 10 seconds within a 30MB file size limit.

Strategy: Optimized IDA* (Iterative Deepening A*).

We will use a smart search algorithm that predicts how far away the solution is (Heuristic). To make this fast enough for Java, we must optimize the memory usage of your existing code.

The Three Pillars

1. **The Engine (RubiksCube.java):** Your physics simulation. *Must be optimized to generate zero garbage.*
 2. **The Brain (ManhattanDistance.java):** The heuristic. *Maps 2D array indices to 3D pieces to calculate distance.*
 3. **The Driver (Solver.java):** The recursive search loop. *Uses IDA* and backtracking.*
-

2 Phase 1: Optimizing The Engine (RubiksCube.java)

Why: Your current code creates new arrays (`new char[]`) every time a move is made. IDA* makes millions of moves. This will crash the Garbage Collector.

2.1 Step 1.1: Change Visibility

Allow the Solver to access the move logic directly.

```
1 // Change from private to public
2 public void applyMove(char move) { ... }
```

2.2 Step 1.2: Add Fast Accessor

Add this method so the Heuristic can peek at colors without copying the whole array.

```
1 public char getColor(int row, int col) {
2     return cube[row][col];
3 }
```

2.3 Step 1.3: Rewrite rotateFace (Critical Optimization)

Replace your loop-based rotation with this "Swap" version. It uses 0 memory allocations.

```
1 private void rotateFace(int r, int c) {
2     // 1. Rotate Corners (TopLeft -> TopRight -> BottomRight -> BottomLeft)
3     char temp = cube[r][c];
4     cube[r][c] = cube[r + 2][c];           // TopLeft = BottomLeft
5     cube[r + 2][c] = cube[r + 2][c + 2];   // BottomLeft = BottomRight
6     cube[r + 2][c + 2] = cube[r][c + 2];   // BottomRight = TopRight
7     cube[r][c + 2] = temp;                 // TopRight = TopLeft (saved)
8
9     // 2. Rotate Edges (TopMid -> RightMid -> BottomMid -> LeftMid)
10    temp = cube[r][c + 1];
11    cube[r][c + 1] = cube[r + 1][c];        // TopMid = LeftMid
12    cube[r + 1][c] = cube[r + 2][c + 1];   // LeftMid = BottomMid
13    cube[r + 2][c + 1] = cube[r + 1][c + 2]; // BottomMid = RightMid
14    cube[r + 1][c + 2] = temp;              // RightMid = TopMid (saved)
15 }
```

Note: Verify the direction (Clockwise vs Counter-Clockwise) matches your specific implementation by testing one face rotation.

2.4 Step 1.4: Fix applyMove Cases

In each case, stop using `char[] temp = new char[3]`. Use variables.

```
1 case 'F':
2     rotateFace(3, 3); // White face
3
4     // OLD: char[] tempF = new char[3];
5     // NEW: Use 3 simple variables
6     char t1 = cube[2][3];
7     char t2 = cube[2][4];
8     char t3 = cube[2][5];
9
10    // ... Perform your swaps ...
11
12    // Restore from variables
13    cube[3][6] = t1;
14    cube[4][6] = t2;
15    cube[5][6] = t3;
16    break;
```

3 Phase 2: The Brain (ManhattanDistance.java)

Concept: We need to map your 2D array indices to "Virtual Cubies".

- **Corner:** A group of 3 indices (e.g., The generic "Corner 0" is composed of the sticker at [0][3], sticker at [3][3], and sticker at [3][0]).
- **Edge:** A group of 2 indices.

3.1 The Code Structure

```
1 public class ManhattanDistance {
2
3     // MAP: Defines which array indices belong to which Corner piece.
4     // There are 8 corners. Each corner has 3 facelets.
5     // Format: { {r1,c1}, {r2,c2}, {r3,c3} }
6     // You must fill these based on your specific layout!
7     private static final int[][][] CORNER_INDICES = {
8         {{0,3}, {3,3}, {3,0}}, // Corner 0: Up-Front-Left
9         {{0,5}, {3,5}, {3,8}}, // Corner 1: Up-Front-Right
10        // ... Fill in the other 6 corners
11    };
12
13    // MAP: Defines which array indices belong to which Edge piece.
14    // There are 12 edges. Each edge has 2 facelets.
15    private static final int[][][] EDGE_INDICES = {
16        {{0,4}, {3,4}}, // Edge 0: Up-Front
17        // ... Fill in the other 11 edges
18    };
19
20    public int calculate(RubiksCube cube) {
21        int totalDist = 0;
22
23        // 1. Calculate Corner Distances
24        for (int i = 0; i < 8; i++) {
25            // Get the 3 colors currently at Corner Position 'i'
26            char c1 = cube.getColor(CORNER_INDICES[i][0][0], CORNER_INDICES[i][0][1]);
27            char c2 = cube.getColor(CORNER_INDICES[i][1][0], CORNER_INDICES[i][1][1]);
28            char c3 = cube.getColor(CORNER_INDICES[i][2][0], CORNER_INDICES[i][2][1]);
29
30            // Identify which physical piece this is (e.g., Red-White-Blue piece)
31            int pieceID = identifyCorner(c1, c2, c3);
32
33            // Look up how far 'pieceID' is from position 'i'
34            // You need a pre-computed table or a smart switch statement here
35            totalDist += getDistance(pieceID, i);
36        }
37
38        // 2. Calculate Edge Distances (similar logic)
39        // ...
40
41        return totalDist / 4; // Divide by 4 is a standard admissible heuristic trick
42    }
43
44    // Helper to identify a piece based on its colors
45    private int identifyCorner(char c1, char c2, char c3) {
46        // Sort characters to make matching easy: "RWB" == "WRB"
47        // Return 0-7 based on the color combination
48        return 0; // Placeholder
49    }
50 }
```

4 Phase 3: The Driver (Solver.java)

Concept: IDA* is a Depth-First Search that has a "Budget". If the cost (moves made + estimated moves remaining) exceeds the budget, we stop (prune).

4.1 The IDA* Implementation

```
1 public class Solver {
2
3     // Moves: Only clockwise. We handle Prime moves logic inside the loop.
4     private static final char[] MOVES = {'F', 'B', 'L', 'R', 'U', 'D'};
5
6     public static void main(String[] args) {
7         // ... File I/O ...
8         // RubiksCube cube = new RubiksCube(inputFile);
9         // String result = solve(cube);
10        // ... Write result ...
11    }
12
13    public static String solve(RubiksCube cube) {
14        ManhattanDistance heuristic = new ManhattanDistance();
15        int threshold = heuristic.calculate(cube);
16
17        while (true) {
18            System.out.println("Searching with max depth: " + threshold);
19            SearchResult result = search(cube, 0, threshold, heuristic, -1);
20
21            if (result.solved) return result.path;
22            if (result.nextThreshold == Integer.MAX_VALUE) return "Unsolvable";
23
24            threshold = result.nextThreshold;
25        }
26    }
27
28    // Recursive Function
29    // 'g': moves so far
30    // 'bound': max allowed cost
31    // 'prevMoveIndex': to prevent redundant moves (like F then F')
32    private static SearchResult search(RubiksCube cube, int g, int bound,
33        ManhattanDistance h, int prevMoveIndex) {
34
35        int f = g + h.calculate(cube);
36        if (f > bound) return new SearchResult(false, null, f);
37        if (cube.isSolved()) return new SearchResult(true, "", 0);
38
39        int min = Integer.MAX_VALUE;
40
41        // Try all 12 moves (6 Clockwise, 6 Counter-Clockwise)
42        // We use a loop 0..5 for clockwise, and logic for primes
43        for (int i = 0; i < 6; i++) {
44            // Optimization: Don't undo the move we just made
45            if (prevMoveIndex == i) continue;
46
47            char moveChar = MOVES[i];
48
49            // --- TRY CLOCKWISE (e.g., F) ---
50            cube.applyMove(moveChar);
51            SearchResult res = search(cube, g + 1, bound, h, i);
52            if (res.solved) return new SearchResult(true, moveChar + " " + res.path, 0);
53            if (res.nextThreshold < min) min = res.nextThreshold;
54            cube.applyMove(moveChar); cube.applyMove(moveChar); cube.applyMove(moveChar);
55            // Undo (3 turns = 1 reverse)
56
57            // --- TRY COUNTER-CLOCKWISE (e.g., F') ---
```

```

57         // "Three Rights Make a Left"
58         cube.applyMove(moveChar); cube.applyMove(moveChar); cube.applyMove(moveChar);
59         res = search(cube, g + 1, bound, h, i);
60         if (res.solved) return new SearchResult(true, moveChar + " " + res.path, 0);
61         if (res.nextThreshold < min) min = res.nextThreshold;
62         cube.applyMove(moveChar); // Undo (1 turn fixes 3 turns)
63     }
64
65     return new SearchResult(false, null, min);
66 }
67
68 // Helper class to return multiple values
69 static class SearchResult {
70     boolean solved;
71     String path;
72     int nextThreshold;
73     public SearchResult(boolean s, String p, int n) {
74         this.solved = s; this.path = p; this.nextThreshold = n;
75     }
76 }
77 }

```

5 Roadmap Checklist

- **Day 1:** Modify `RubiksCube.java` (Phase 1). Verify `applyMove` works by writing a small test (Apply F, Apply F', assert solved).
- **Day 2:** Map the indices. Fill in the `CORNER_INDICES` and `EDGE_INDICES` arrays in `ManhattanDistance.java`. This is the hardest part—be careful with the numbers!
- **Day 3:** Write the `Solver.java` logic. Start testing with very simple scrambles (1 or 2 moves).