# Java Rubik's Cube Solver

Armin Ahmadi | 301610827 | aaa360@sfu.ca

Nathan Omana | 301596662 | nao3@sfu.ca

Simon Fraser University

Fall 2025

**Author Note**

This document illustrates Armin and Nathan's development process and workflow throughout the lifecycle of this Rubik's Cube solver. In this paper, we will take you through our initial project timeline and describe the stages of constructing the solver. These stages include our resources, the many failures of our code, as well as the improvements that led to our finalized successful approach.

**Project Timeline: (December 2, 2025)**

*Armin*
*Nathan*
*Armin and Nathan*

| **Task:** | **Complete by:** | **Progress:** | **Notes:** |
|---|---|---|---|
| Gather resources and research how to intuitively solve a cube. | Nov 1, 2025 | Completed | Resources in the resources section |
| Implement Cubie Model in Cube.java | Nov 3, 2025 | Completed | |
| Double check move mappings are accurate (Add debug functions in Cube.java) | Nov 5, 2025 | Completed | |
| Begin implementing BFS in Solver.java | Nov 12, 2025 | In Progress | BFS implementation successful but yielded incorrect solutions. (Check cubie model mappings) |
| Revise cubie mappings and push to git | Nov 12, 2025 | Completed | |
| Test and debug BFS implementation with working cubie model | Nov 13, 2025 | Completed | Initially can only solve solution 2(FBBB) <1 second.<br><br>- After revising and adding pruning, (next move cannot be redundant ie. undoing a move)<br>It solves the first 3 under 15 seconds. Still times out for 4-10 (potentially running out of memory?) |
| Implement IDA* for more complex scrambles | Nov 16, 2025 | In Progress | **Unsuccessful -** can only solve 1-4 in around 30 secs. |
| Determine if heuristics are inefficient | Nov 20, 2025 | Completed | Calculating edge and corner misplacement falsely assumes that a move that fixes one case automatically fixes the |

| | | | |
|---|---|---|---|
| | | | other, causing bad moves to be admissible. Should we try another method?<br>+ It might be faster to have less constraints with heuristics and just yield a longer result. |
| Instead of position-based heuristics, check whether a certain move brings the cube state closer to the "cross-position" -> check human based algorithms in resources | Nov 23, 2025 | In Progress | Switching to a layer-by-layer approach that checks a cube's progression towards being solved based on human heuristics yields long solutions but all around 45 seconds which is even better than the pdb approach. |
| Implement move pruning to find a solution faster | Nov 25, 2025 | Completed | Continuously cloning the cube makes the search process really slow. I tried a recursive approach with a single cube instance instead and tweaked the heuristics a little bit to be more lenient on moves (check git). |
| Test move validity manually and with a python script to ensure they are all correct | Nov 27, 2025 | Completed | All solutions are correct, but 5,8,21,27 still take around 30 seconds. |
| Trial and error heuristic tweaks to check whether time complexity is reduced | Nov 30, 2025 | Completed | Make heuristics trade a longer solution for a faster response -> check documentation. |

**<u>Documentation Outline:</u>**

***Draft: Cubie Model + BFS***

Idea: The Cubie model would reduce time and space complexity for BFS, ie, checking corner and edge positions rather than ‘wrong sticker’ positions.

We used four 1D arrays to keep track of the corner and edge positions (permutations) to map 20 physical pieces instead of the 54 in the facelet model. However, to achieve this, we had to first use the facelet model and then map those to the cubie, and we did this in the Rubik's Cube constructor, which proved to be quite inefficient in retrospect. For this BFS implementation, we used a queue that parses the current cube state and then applies all 6 rotations to generate new cube states. This approach guaranteed the shortest possible combination of moves for the solution, but increased the space complexity drastically as it explored every move 1 step away. This implementation of BFS grew exponentially, causing it to fail past scramble02.

After researching how to optimize our BFS approach, a bidirectional BFS seemed more promising as it would cut our space/time complexity in half. We used a forward and backward queue to explore cube states from the scrambled and solved state (using the same searching knowledge in the first rendition). The logic behind it is:

- Check if the move mappings already exist in the backward queue mapping; if not, then move forward once and move backward once
- If they do exist, a collision condition is met, meaning there is a solution found.

This bidirectional approach solved scrambles (01-03) in around 15 seconds, but timed out past scramble03.

To solve cubes more efficiently, we decided to take a heuristic-based (IDA*) approach (Kociemba’s Two-Phase Algorithm), as exhausting all scramble possibilities was not feasible given the 10-second time constraint. Referring to “Writing Code to Solve a Rubik’s Cube” by Brad Hodkinson, he illustrates how there exists a ‘G1’ state where all corners, edges, and slice edges are positioned correctly. From this G1 state, the next moves to solve the scramble consist of “<U, D, F2, B2, R2, L2>”. In theory, using this algorithm would greatly reduce the number of nodes the search must traverse in addition to solving scrambles faster.

To implement this into code, within our Solver.java, we utilized many helper classes structured around one instance of the cube. ‘applyPhase1’ applies a clockwise turn on the cube and calculates the heuristic that checks whether the move brings a corner/edge into the correct positioning. Still, this method proved to be too slow as it explored too many branches and timed out during the testing of all cases. Through the RedBlobs forum found in the resources list, we explored the idea of adding a greedy multiplier that multiplies the heuristic value by 3 in order to aggressively prune more of the misplaced edges/corners. This yielded solutions for scrambles01 and 02 in around 10 and 5 seconds, respectively, but timed out from 03-10. This is when we decided to switch our heuristic approach to yield faster results, as we referred to Piazza and realized that finding an optimal solution was not necessary.

## ***Current Final Model***

Idea: The final approach to our solution algorithm proved to be a huge pivot from pure BFS techniques (what we initially started off with). After realizing that an optimal solution was not necessary, we had the idea to approach the problem using a phase reduction algorithm, similar to what we use in real life to solve a regular 3x3 Rubik's Cube. We used Jperm’s CFOP algorithm as well as the 2 Look versions of the PLL and OLL algorithms (found in the resources section). Jperm’s tutorial and website on how to solve a real Rubik’s Cube was our main tool in addition to the other sites mentioned in the resources part of the documentation. We used these algorithms for both the macros and the overall logic used in this solution. We combined this strategy with IDA* and PDBs, which yielded success.

To solve the cube, we address it in 7 distinct, easier-to-solve problems. *The cross,* which solves the down face edge pieces. *The Corners,* which solve the down face corner pieces. *Middle Edges (F2L),* which solves the middle layer edges. *Edge Orientation,* which orients the up face edges (Top cross). *Orientation of the Last Layer (OLL),* which orients the Up face corners (top face). *Corner Permutation,* which permutates the up face corners. *Edge Permutation,* which permutates the up face edges and finally solves the cube.  Additionally, we set up our PDB tables (*edgeDistTable*) and (*cornerDistTable*) and initialized them using a BFS on the simplified piece space to pre-calculate the minimum moves from every possible state of a single piece (position + orientation), all the way to its solved state. We also utilized predefined macros (eg, Sune, U-Perm, refer to code for more). These moves are complex sequences that are treated as a single move, which allows the search of OLL/PLL algorithms in one step rather than us generating them, which speeds up the process. This logic, paired with IDA*, worked for us and resulted in solving all but four cases in which the search timed out, although it would take, on average, about 45 seconds to solve one scramble. We then optimized the code using four techniques, which resulted in our final optimized code. Our first optimization was to replace our old method, where we created a deep copy of the state array for every node explored in the search (This was done by using *Cube.clone()*). We implemented a recursive backtracking logic. We created a function called *performInverseMove* in our cube.java, which, when put simply, restores the "cube"'s object's state to what it was before the move was applied. This ensures that when we move onto the next move, it starts from the correct, unmodified parent state. Our second optimization was to prune the same move four times, as this would be equivalent to doing nothing (four moves on a Rubik's Cube puts you back exactly where you were) (*i == lastMove && consecutiveMoves >= 3* on line 164 of *find.java*). Our third optimization technique was to avoid redundant commutative move sequences like *"DU"* if *U* was our last move, as DU and UD are commutative and result in the same cube state, so exploring both UD and DU would be redundant. Our last optimization strategy was to change the weight of our heuristic; our cost was calculated by "*f = g + h(n) * 2",* where the heuristic was multiplied by 2. We tried changing this with different numbers like 3,4,5, and found 3 to work the best through trial and error. This multiplier makes our heuristic greedy by pushing the search toward states that reduce our heuristic cost (*prunes our long and low-impacting branches*). All of these optimizations, combined with our IDA*, PDBs, Macro and Human algorithm logic, were able to give us our final code, where we solve 36/40 of the scrambles under 10 seconds and solve all under 18 seconds when tested on CSIL computers.

## Resources

https://rubiks-cube-solver.com/ -> used this for testing scramble solutions

https://www.youtube.com/watch?v=YsWKrxAbopk -> explains how any scramble can be solved in an upper bound of 20 moves ('God's Number')

https://www.geeksforgeeks.org/dsa/bidirectional-search/ ->How to implement bidirectional BFS

https://www.speedsolving.com/wiki/index.php/Kociemba%27s_Algorithm -> promising two-phase ida* approach

https://medium.com/@brad.hodkinson2/writing-code-to-solve-a-rubiks-cube-7bf9c08de01f -> various algorithms using ida* to solve a 2d or 3d representation of a cube (array representations)

https://solvethecube.com/ -> human based algorithms

https://www.redblobgames.com/pathfinding/a-star/introduction.html -> greedy ida algorithms

https://softwareengineering.stackexchange.com/questions/365497/how-to-measure-the-solution-level-of-an-unsolved-rubiks-cube -> reference for computing manhattan distance formula

https://jperm.net/3x3/cfop -> Jperm's CFOP (Cross, F2L, OLL, PLL) tutorial

https://jperm.net/algs/2look/oll -> Jperm's 2 look OLL -> Sune Algorithm and I-Shape

https://jperm.net/algs/pll and https://jperm.net/algs/2look/pll -> Jperm's PLL and 2 Look PLL algorithms

https://semmi88.github.io/blog_posts/rubiks_cube.html#:~:text=And%20in%20fact%20this%20non%2Dcommutative%20property%20is,then%20solving%20the%20cube%20would%20be%20trivial. -> where we learned the commutativity rule of rubik's cube