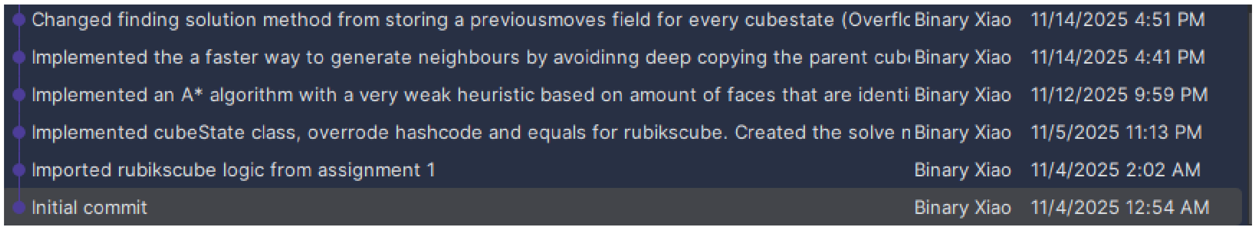
Binary Xiao

Initial Approach (Excluded from final solution)

My first approach to this project was to develop an A* algorithm and tweak the heuristic until it was advanced enough to solve a cube. At this time, I didn't have much of a plan but I thought this was a standard solution for this project,



```
Changed finding solution method from storing a previousmoves field for every cubestate (Overflc Binary Xiao   11/14/2025 4:51 PM
Implemented the a faster way to generate neighbours by avoidinng deep copying the parent cub Binary Xiao   11/14/2025 4:41 PM
Implemented an A* algorithm with a very weak heuristic based on amount of faces that are identi Binary Xiao   11/12/2025 9:59 PM
Implemented cubeState class, overrode hashcode and equals for rubikscube. Created the solve n Binary Xiao   11/5/2025 11:13 PM
Imported rubikscube logic from assignment 1                                                  Binary Xiao   11/4/2025 2:02 AM
Initial commit                                                                               Binary Xiao   11/4/2025 12:54 AM
```

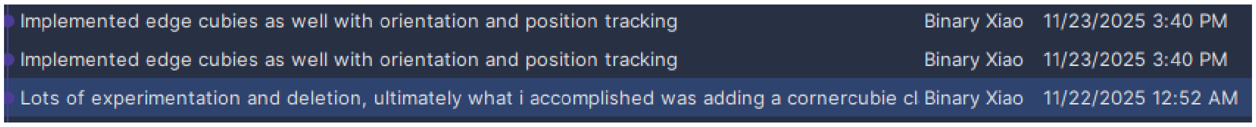
I spent about 10 days setting up the foundations for this plan and implementing it with a basic heuristic of comparing every sticker face to a solved state and adding a point if it didn't match. Rather than just use the RubiksCube class from assignment 1, I created a new class cubeState that could wrap the cube with other necessary methods and attributes. cubeState contained the RubiksCube, the heuristic score for each cube during A*, the move that brought it to that state from its parent, and several methods including compareTo, equals, hashCode, getNeighbours (generated all 18 of the move neighbours for the cube and returned a hashset of them) and getHeuristic (method calculating heuristic then storing it within the object).

The A* algorithm I used was based entirely off the algorithm shown in the class slides. I placed all queued states in a priorityqueue, with the cubeState carrying the lowest heuristic score + distance from start being prioritized. Distance was tracked within the A* algorithm by initializing each state's distance as current.distance + 1. Looking back, this could have been included in the getNeighbours method.

With this basic setup, I managed to solve scrambles 1-3 but no luck with anything else. I spent the rest of the first 10 days focused mostly on optimizations to avoid memory overflows and then tried tweaking the heuristics but no luck. I was achieving faster solves on scrambles I could solve, however.

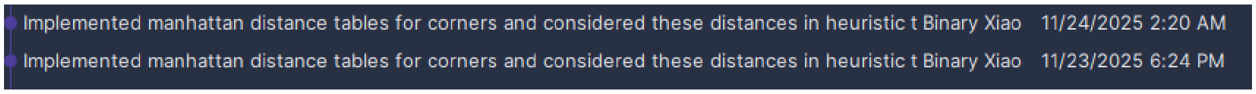
Intermediate approaches (Exluded from final solution)

After this, I entered the most frustrating part of the project which was trying to implement a working heuristic with the preliminary goal of solving scramble 4. Because I spent a lot of time in this stage reverting to my previous git commit after failed attempts, I don't have many of the specific things I tried documented. Generally, my strategy revolved around counting solved corners and edges as that was the only obvious starting point. I first started with checking how many corners were solved, then edges, then added orientation by giving a more severe heuristic penalty if corners/edges were unoriented. Since I had a sticker representation this was quite complicated to implement at first, and I later created corner and edge cubie objects within my cubeState class so I could create collections of them for each cube and assign unique attributes. These classes stored and calculated a piece's original position, current position, orientation, and stickers.



```
Implemented edge cubies as well with orientation and position tracking              Binary Xiao   11/23/2025 3:40 PM
Implemented edge cubies as well with orientation and position tracking              Binary Xiao   11/23/2025 3:40 PM
Lots of experimentation and deletion, ultimately what i accomplished was adding a cornercubie cl Binary Xiao   11/22/2025 12:52 AM
```

During this process, I discovered my move application in my assignment1 RubiksCube had a small error in one of the moves causing issues in my corner and edge cubies and fixed it. Using my new cubies, I moved onto more advanced heuristics, most notably one that tracked a cubie's distance in moves from its original position. I created 2D tables for edges and corners using a BSF for each possible combination of two cubies on a solved cube to find the minimum moves to move from one corner/edge position to the goal corner/edge position. Using these tables, the times for scramble03 improved significantly from about 3000ms to 450ms, but still could not solve scramble04.



```
Implemented manhattan distance tables for corners and considered these distances in heuristic t Binary Xiao   11/24/2025 2:20 AM
Implemented manhattan distance tables for corners and considered these distances in heuristic t Binary Xiao   11/23/2025 6:24 PM
```

Some of the main issues were that I could not reliably get an accurate estimate of moves from the current state to solved because I was giving penalties to corners and edges individually, even though they are all connected and one move may disrupt multiple or solve multiple. Furthermore, upon making the table I realized with all 18 possible moves available, most cubies are only 1 step away from a goal state and some are at most 2 steps away. This meant that the distance between cubies was not a very helpful sign for solvability.

From this point, I began to explore options outside of A* because I was running out of ideas and there was very little documentation online about Rubiks Cube solving programs that used A*. Furthermore, I was very reliant on intuitive things like similarity to a solved state because I have no idea how to solve a Rubiks Cube myself. In fact, all the ones I found online were based on using pattern databases, IDA* and pre existing solve algorithms like Kociemba's two phase or Korf's algorithm. However, it's important to note that these were designed for optimal solving, unlike our assignment.

Final Solution

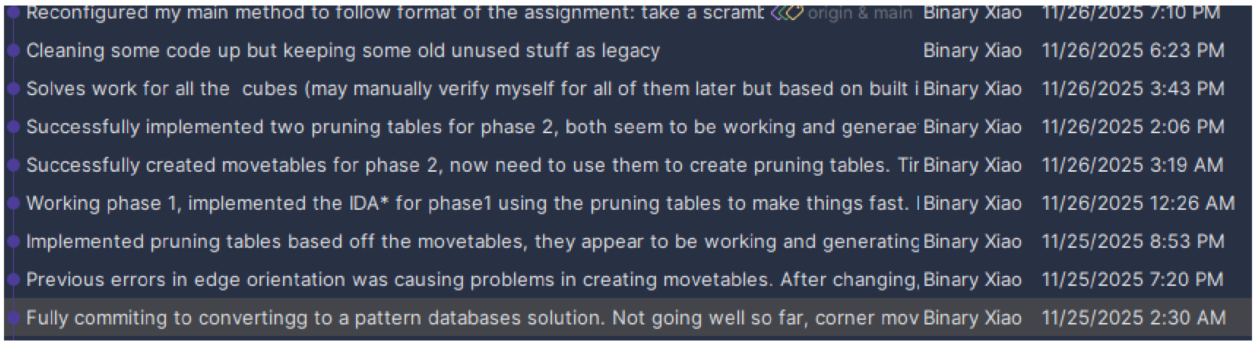
After exploring pattern databases, I attempted a more basic version by indexing corner and edge permutations into integers which acted like codes for the cube's state and created a table that tracked how far each state was from a solved state. However, I could only use half the edges or it would take much too long to generate. This idea fizzled out quickly as I saw no results from it.

After lots of research and reading, I committed to a Kociemba 2-phase algorithm, these pieces of documentation were very helpful to guiding my solution.

https://kociemba.org/math/twophase.htm

russfeld.me/projects/rubiks/presentation.pdf

The final commits are all focused on this method and brought me to a working program.



```
Reconfigured my main method to follow format of the assignment: take a scramb  origin & main  Binary Xiao   11/26/2025 7:10 PM
Cleaning some code up but keeping some old unused stuff as legacy                             Binary Xiao   11/26/2025 6:23 PM
Solves work for all the  cubes (may manually verify myself for all of them later but based on built i Binary Xiao   11/26/2025 3:43 PM
Successfully implemented two pruning tables for phase 2, both seem to be working and generae Binary Xiao   11/26/2025 2:06 PM
Successfully created movetables for phase 2, now need to use them to create pruning tables. Tir Binary Xiao   11/26/2025 3:19 AM
Working phase 1, implemented the IDA* for phase1 using the pruning tables to make things fast. I Binary Xiao   11/26/2025 12:26 AM
Implemented pruning tables based off the movetables, they appear to be working and generating Binary Xiao   11/25/2025 8:53 PM
Previous errors in edge orientation was causing problems in creating movetables. After changing, Binary Xiao   11/25/2025 7:20 PM
Fully commiting to convertingg to a pattern databases solution. Not going well so far, corner mov Binary Xiao   11/25/2025 2:30 AM
```

To sum up the idea, the cube undergoes a transformation called phase 1 to a goal state where corner orientation, edge orientation and middle edges are all in the middle slice (same as solved). From this state, only the moves U, D, FF, BB, RR, LL, UU, DD, UUU, DDD will be used to get to the solved state as they do not affect any of the above.

First I created indexers to transform all possible edge orientations, corner orientations and permutations of the middle edges (ignoring order) into integer values which acted like coordinates. I followed the formulas in official kociemba documentation for this. Then, based on these tables I created movetable generators (one each for edge orientation, corner orientation and middle edge) which ran BFS on a solved state and tracked how each move would affect corner orientation/edge orientation on a state. This was implemented in the form of a 2d array where each row was a unique orientation, and each column was one of the possible moves that could be applied. The value stored as an integer was the resulting orientation after the moves was applied. I encountered a major issue here where the edge orientation movetable would not fill completely. This was very difficult to debug as there were many reasons why this could be going wrong (Rubikscube move logic, wrong indexing, wrong orientation logic in cubies etc.) and I spent an entire day on this. I realized after going through all my code it was because I was using the wrong definition of edge orientation (facepalm). I rewrote the code and stuck to this definition from the speed solving wiki:

- If the sticker has L/R color it's a bad edge.

- If the sticker has F/B color, look at the sticker on the other side of the edge.

* If the side sticker has U/D color, it's a bad edge.
After successful movetables, I created pruning tables with BFS from solved state again which were generated very quickly using the movetables (Only need to perform an array lookup to see what the next state is). These pruning tables would store how far away a combination of corner orientation with middle edge permutation or edge orientation with middle edge permutation (for two tables total) was from the phase 1 goal. For example, table[333][67] would return the distance a cubestate with corner orientation of 333 (or edge orientation) and middle edge permutation of 67 was from the solved state. Since one table combining all 3 would be too large, I opted to take the maximum value returned between the two for my heuristic rather than track all possible combinations of the three (Since if at least 8 moves are needed to fix corner orientation, then it doesn't matter if fixing edge orientation only takes 3, the minimum moves required is 8). For the actual process of solving, I used IDA* search that was based off the heuristic provided by my pruning tables. The goal state was a cube state where all the 3 different coordinates were at the "correct value" which were all calibrated to be 0.
after all tables were generated, the process of getting to phase 1 was extremely fast because of how precise the tables were.

For phase2, I did a very similar process but with different coordinates. To get from phase 1 to solved, I had to track corner permutation, edge permutation (Only 8 because 4 are the middle edges already in the correct slice), and the ordering of the middle edges within the middle slice. I once again used formulas from official kociemba documentation to index the three different coordinates. Then, I used these coordinates to create three movetables, and these movetables were used to generate the pruning tables. Once again to avoid the table being too large, I created two by combining corner permutation and edge permutation with middle slice ordering and took the max distance returned by the two for my heuristic. It is important to note that generating tables in this phase took much longer (about 3 seconds total) because of the huge number of possible permutations. For the solving process, I took the phase 1 cube and applied an almost identical IDA* search algorithm to it. This time, the goal state was a solved state and only the "safe" moves of U, D, FF, BB, RR, LL, UU, DD, UUU, DDD were explored.

The heuristic from using this pattern database solution was strictly superior to my previous attempts of counting stickers, tracking misaligned pieces or orientations, and manhattan distances. I managed to solve all 40 cubes, most of them in under or just about 10 seconds, while some of the harder ones took up to 35 seconds including table generation times.

My closing thoughts about this project are that it was extremely difficult, and definitely the hardest school assignment I've ever tried before. The most time-consuming aspect of this project was trying to debug/improve my different heuristics. It was very difficult to pinpoint why exactly a heuristic wasn't working that well because I wasn't quite sure what exactly needed to be achieved (Cubes can look solved but need to be deconstructed again to be solved). That's part of why I opted for the pattern database solution even though I was concerned about overcomplicating things, since it gave exact distances without needing to try and estimate based off a cube's current state. I wouldn't say any part of this project was easy, it took me a very long time just to understand the concepts and mechanisms I needed to implement before I even began coding (Cubies vs facelets, what is cube orientation, what is a pattern database/movetable etc, what algorithms are used to solve cubes naturally etc.).

Below I've attached my solution to every cube

```
cube 0: Solved! Solution: URUUURRR
cube 1: Solved! Solution: BBBFFFFF
cube 2: Solved! Solution: UUURRRUUURRRUUUFFF
cube 3: Solved! Solution: RRRLDUUUBBBFFFUULLDDRRUULLBBRRLLUUUD
cube 4: Solved! Solution:
RDBBLDDDBBBUUUDDDLDDDFFDDBBDDDRRBBURRUUUBBUULL
cube 5: Solved! Solution: FFDFLBBBDDDRDLLBRUUUBBDFFRRLLURRDDBBUUURRDLLRR
cube 6: Solved! Solution: BBBFRRRDDUUUBBRUUFRRRUBBLLDDDBBDUFFRRUULLD
cube 7: Solved! Solution:
LLDDDLLLUUBBBRRRFFLLLDDBUUUFFDDRRBBUUULLRRFFUUURRBBUFF
cube 8: Solved! Solution: UUBBBLULLBBDDFFFRRUUBBLLUUUBBDLLBBDDFFULLD
cube 9: Solved! Solution: DDBBFFFDRLLFFRBRRBBUUUFFDUULLDRRFFDDLLFFRR
cube 10: Solved! Solution: RRRRR
cube 11: Solved! Solution: FFFFF
cube 12: Solved! Solution: LLBDDBB
cube 13: Solved! Solution: BBBRRBLLL
cube 14: Solved! Solution: LLBDDDFFFFF
cube 15: Solved! Solution: RRBLLLFBFFDDRRFFDDBBLLFFUU
cube 16: Solved! Solution: DDDUUUFFFDDDFFRFFF
cube 17: Solved! Solution: FFUUURRRULLLFFFFFU
cube 18: Solved! Solution: FFFUUFRRRFFFRURRFFUUURRUFFRRUUUFFUULL
cube 19: Solved! Solution: BRRRRRURBBBRRUUUBFFBBRRDFFUBBFFDDDRRBBD
cube 20: Solved! Solution: RLFFDDUBRULLFFF
cube 21: Solved! Solution: RBBDRRLLLBBB
cube 22: Solved! Solution:
BBBLLURRUUBBBRRUDDDLLLUUUFFDDBBLLUFFDDDDDFFDDDLLRR
cube 23: Solved! Solution: FFFRRRDLLFRRDDRRRLLBBUUUULLDDRRFFLL
cube 24: Solved! Solution: DDRRFFFDDRRRFRRBBLLFFRRDLLUUUUUFFULL
cube 25: Solved! Solution:
LURRRLLLBUUUDBBFFFRRRDDBBLLDDDRRUUDLLUBBUURRUUUFF
cube 26: Solved! Solution: FFFBBBRFUUUBBBLDDDUUUBUFFDLLUFFDLLDDDRRBBUUU
cube 27: Solved! Solution:
FUUURRDDFUUULLDDLBBBDRRUULLBBDFFRRDDLLUUUFFLLU
cube 28: Solved! Solution: FBBRLLBBUUDDRR
cube 29: Solved! Solution:
LLBLLLUUUDDDRRRUUURRFFFFFBBUUULLRRFFUULLUBBDDDFFBBDD
cube 30: Solved! Solution:
LDRRRBFFFDRLLDLLLRRDDDLLBBUUUFFURRDUURRBBLLRRDD
cube 31: Solved! Solution: FFFRRDRLLLFFFUFFUUURRUFFRRDDDLLRRDDDBBUU
cube 32: Solved! Solution: LFFFDDDRLLLBBDFFFUDDFFLLUUFFLLDFFUUUBBRRDDD
cube 33: Solved! Solution:
RRRFFUUUBLDFBBURRRLLDDDRRDDBBFFLLUUUFFDLLDDRRUUU
cube 34: Solved! Solution:
DDDLLBUUDDDRUFFFDRRRDDFFDLLDDBBLLDRRBBDDDRRUUUBB
cube 35: Solved! Solution: DFLFFFFFDDDRRRBBBUUFULLFFDDLLDDDFFDRRDDLLBBD
cube 36: Solved! Solution:
RRRLBBBDDDUUUFLFFFUULFDDRRBBDFFULLDDDDDLLUURRUFFUU
cube 37: Solved! Solution:
UUUFFFUURRRDBBRDDDLFFFUUDBBULLFFRRLLDDRRUUUBBRR
cube 38: Solved! Solution:
BBBFFFDBBBUULLLFFUUUFBBDDDLLDDBBUFFLLUUURRDDLLUUU
cube 39: Solved! Solution: LLLBBBDFRLLFBBRRRFFDDDLLUUUDDBBLLDFFUUULLDDDFF
```