Ruvi Muskwe
Shahar Bano

Rubiks Cube Final Project Documentation

1) Our Thought Process-

In the initial week following the assignment's release, we commenced our research into the algorithms we intended to utilize. Our preliminary consideration was the Iterative Deepening A* (IDA) algorithm, as we believed its space efficiency would facilitate the quickest solutions, although we were uncertain regarding the appropriate heuristic to accompany it. Nevertheless, the strong endorsement of the A* algorithm by Professor Shinkar during weeks 8 and 9, along with the TAs, significantly influenced our planning process. Despite encountering initial challenges with A*, we ultimately decided to proceed with it, concentrating our efforts on formulating an effective heuristic. Once we finalized the algorithm, we established three fundamental principles to guide our implementation:

1. The Rubik's Cube solver must be memory-efficient
2. It should aim to solve cubes within ten seconds
3. It needs to be robust enough to handle challenging, deeply scrambled cubes, not just simple test cases

2) The Algorithms We Tried-

During our development phase, we examined a range of search algorithms. The primary algorithms we opted to test were 2-phase, IDA*, and A*. Although we also considered BFS, we ultimately decided against it after determining that our solution did not necessitate the shortest path.

Initially, we attempted to utilize the A* algorithm; however, we quickly abandoned this approach due to the excessive time it required to resolve the more complex scrambled test cases, coupled with our inability to identify an effective heuristic. **(Figure 1)**

Shortly thereafter, we contemplated a complete overhaul of our code and decided to implement IDA*, which proved to be significantly more challenging than A* and took us several days to accomplish. We found it to perform adequately, and we endeavored to integrate IDA* with concepts from two-phase solvers **(Figure 2)**. The intention was to employ IDA* for the primary search while allowing two-phase principles to provide guidance. Unfortunately, our IDA* implementation and heuristic lacked sufficient strength, resulting in the repeated generation of identical states and a considerable slowdown. Additionally, the two-phase methodology relies heavily on advanced cube mathematics, such as group theory and pattern databases, which we had not yet mastered, leading to a lack of confidence in our ability to implement it. Ultimately, we were unable to successfully merge the two strategies, and as a result, we did not pursue this avenue further.

In conclusion, we revisited the A* algorithm for a second attempt, as it had previously demonstrated the most promise. We returned to A* and ultimately committed to this algorithm because it was the one we comprehended most thoroughly, and it exhibited stability and consistency, even with variations in heuristics.

We retained the earlier implementation from our first attempt **(Figure 3)** but modified our heuristic accordingly. Once we developed a functional version, it became easier to comprehend, debug, and enhance our code.

3) What We Tried That Worked And What Didn't Work-

Throughout the duration of the project, we continuously refined our methodology as several initial concepts either hindered the solver's performance or utilized excessive memory. Even though some peers suggested switching away from A*, we decided to continue using it. The professor and TAs strongly recommended A*, and it offered a clearer structure for us to build on compared to the alternatives we tested. The main difficulties came from making A* actually usable, which depended entirely on having a heuristic strong enough to guide the search.

i) Heuristic Development

Early on, we tried multiple heuristics and algorithms, but most were either too slow or too memory-intensive. Our simplest attempt, heuristic 0, compared each sticker to its face center and counted mismatches. It was easy to implement, but barely guided the search. A* expanded huge numbers of states and often timed out.

We also experimented with IDA* with pruning, IDA* using corner and edge heuristics, a basic DFS, and A* with our first custom heuristic. We even attempted IDA* with a two-phase approach and small pattern-database experiments, but the memory usage was far too high. Our peers suggested a meet-in-the-middle strategy using bi-directioanl A*, but that also required data structures larger than what the assignment allowed.

Most of our time went into exploring ways to strengthen the heuristic without needing large lookup tables or heavy preprocessing. We tested ideas like solving corners first or designing separate heuristics for edges and centers, but none of these alone performed consistently well. Building something that balanced strength, speed, and memory became the core challenge.

ii) Challenges

Creating a useful heuristic was by far the hardest part of the project. Representing the cube and implementing the move functions were manageable, but developing a heuristic that was informative, fast, and memory-efficient required many rounds of trial and error. Many ideas initially seemed promising, but they failed under testing because they either didn't prune enough, expanded states too slowly, or were too weak to find a solution within the limited time frame. At the same time, a heuristic that was too "strong" could strain memory, and overestimating the heuristic risked causing the search graph to grow exponentially. Therefore, balancing strength, efficiency, and memory usage was critical.

iii) Abandoned Approaches

IDA with pruning*
IDA* initially seemed like a good alternative to A*, as it theoretically reduces memory usage by performing depth-first searches iteratively with increasing cost limits. We

combined it with pruning strategies to try to reduce unnecessary state expansions. While IDA* worked well on simple scrambles, it struggled significantly with more complex or deeper scrambles. The heuristics we designed were not strong enough to guide the search efficiently, so the algorithm ended up exploring a massive number of states. Pruning could have mitigated this problem, but implementing effective pruning required additional memory to track already visited states or subpaths. This negated the memory advantage of IDA* and made the algorithm less practical than A*.

Two-phase reduction

Theoretically, this approach can drastically reduce the search space. However, turns out, correctly implementing it requires understanding group theory, coordinate systems, and specialized move reductions. Our partial understanding of these concepts made implementation risky; an incorrect reduction could result in unsolvable states or incorrect solutions. Given the time constraints and the risk of fundamental mistakes, we decided not to pursue this approach further.

Pattern Databases

Pattern databases are precomputed tables that store the optimal moves required to solve specific sub-problems of the cube, such as edge or corner positions. In theory, they provide very strong heuristics for algorithms like A* or IDA*. However, we encountered significant practical problems with pre-computation. Generating these tables took a long time, and storing all possible states in memory proved highly inefficient. Even when we attempted a smaller edge-based pattern database to save memory and reduce preprocessing time, it still required substantial memory and runtime to compile. Creating larger or full databases at runtime was completely infeasible because it would exceed the assignment's time constraints. We also attempted to store the database externally and load it dynamically during execution, but this approach failed due to memory limitations and integration challenges. Overall, the tradeoff between memory usage, preprocessing time, and runtime performance made pattern databases an impractical solution for our project.

4) Our Final Verdict/Implementations And What We Ended Up Keeping-

Our final implementation comprises two primary classes: Solver.java, which orchestrates the A* search algorithm, and CubeState.java, which encapsulates the cube's data structure and movement logic. Initially, we considered introducing a separate Node class to manage search states, but we ultimately integrated this functionality directly within the Solver to maintain conceptual clarity and reduce architectural complexity. For the internal representation of the cube, we adopted a three-dimensional character array (char[][][]), which provides an intuitive, spatially faithful model of the puzzle. While this approach incurs a performance penalty due to the necessity of deep-copying the array for each newly generated state, consultation with teaching assistants affirmed that transitioning to a more memory-efficient flat array would necessitate an extensive and error-prone refactoring, potentially yielding thousands of additional lines of code. Consequently, we prioritized maintainability and comprehensibility over marginal gains in execution speed, a deliberate trade-off that facilitated more straightforward debugging and collaborative development.

1. *Cube Representation – Ruvi/Shahar*

The selection of a 3D array was the product of careful consideration. Although alternative representations were explored, the spatial alignment of the 3D structure with the physical cube offered significant intuitive advantages during the implementation and verification phases. The principal drawback—the computational overhead of replicating the entire state for each node expansion—was a known concession. To mitigate complexity, we implemented a system of fixed face indices, eliminating the need for runtime reorganization of the cube's coordinate mapping. While this design is not optimized for peak performance, its transparency and ease of interpretation were deemed paramount, enabling efficient identification and resolution of logical errors throughout the development cycle.

2. Move Implementation – Shahar

The mechanical operations of the cube were implemented as a set of discrete, face-specific rotation functions. After the clockwise rotations for each face were meticulously validated, the codebase was treated as a stabilized foundation; subsequent

modifications were avoided to prevent the introduction of subtle, cascading errors. Counter-clockwise (prime) and 180-degree (double) moves were consequently derived through the repeated application of the base clockwise rotation—a method that, while not computationally optimal, guarantees correctness through its reliance on already-verified logic. This conservative strategy ensured operational reliability and simplified the reasoning process for us.

3. Final Heuristic – Ruvi/Shahar

The development of an effective heuristic constituted the most intellectually demanding phase of the project. Our final heuristic operates by enumerating misplaced corner and edge cubies, assigning a greater weight to corner mismatches due to their higher constraint satisfaction cost. This heuristic, while not as theoretically powerful as pattern databases or group-theoretic metrics, represented the most sophisticated function we could confidently implement and verify for admissibility. Its construction required significant iterative testing to ensure it provided meaningful guidance to the A* algorithm without inducing overestimation or misleading the search trajectory.

4. Pruning – Ruvi

To enhance the efficiency of the A* search, a series of pruning rules were incorporated directly into the state expansion logic. These rules prohibit redundant or clearly suboptimal maneuvers, such as applying a move that directly reverses the preceding action, or immediately transitioning to the opposite face of the cube. These constraints, though simple in formulation, yielded a dramatic reduction in the effective branching factor, directing the search algorithm toward more productive regions of the state space and conserving valuable computational resources.

5) Closing Remarks-

Although our solver did not identify a solution for every conceivable scramble within the imposed constraints, the project served as a profound case study in the practical trade-offs between algorithmic efficiency, memory utilization, and implementation complexity.

More than a mere academic undertaking, it provided a tangible demonstration of how theoretical principles from heuristic search and combinatorial optimization can be synthesized into a functional application capable of solving a non-trivial real-world puzzle. The experience ultimately reinforced a critical lesson in software development: a robust, well-understood, and maintainable solution frequently holds greater practical value than a theoretically optimal but untenable or incomprehensible one.

Figures/Screenshots From Our Code On Github-

Figure 1:

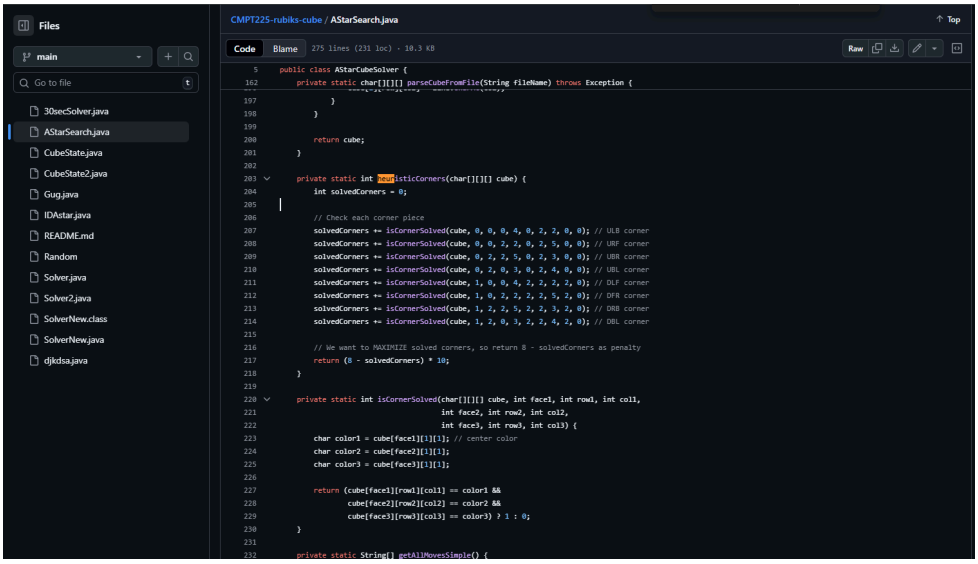


Figure 2:

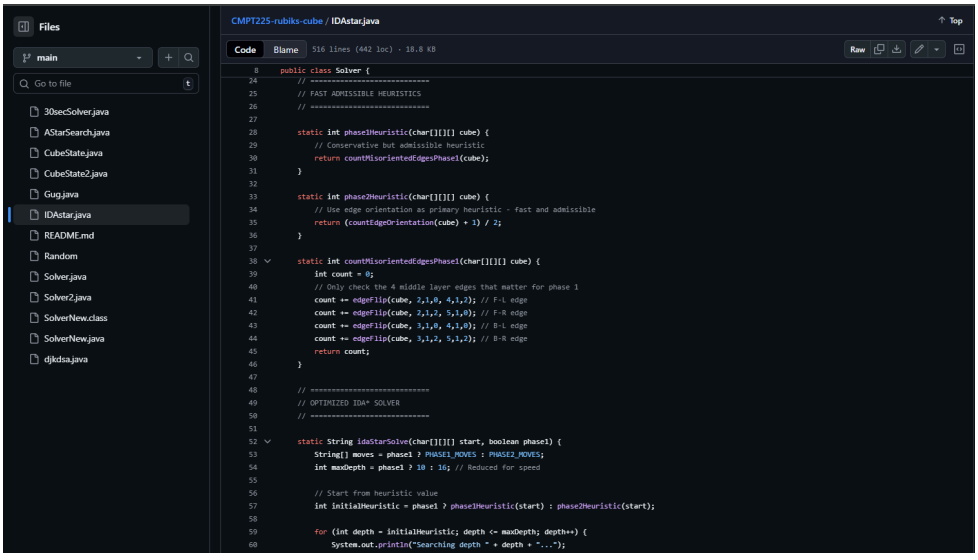
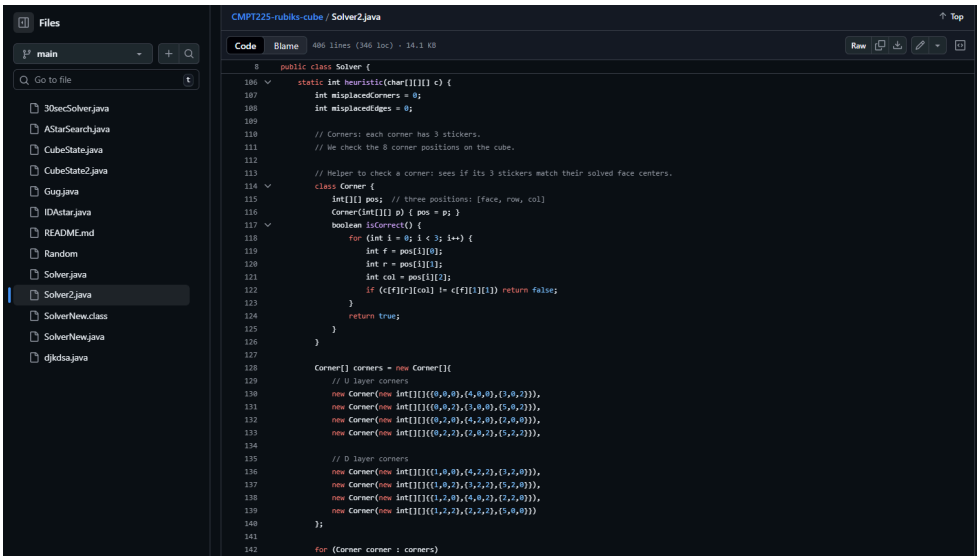


Figure 3:



Sources From Our Research-

- PDBs: https://www.cs.princeton.edu/courses/archive/fall06/cos402/papers/korfrubik.pdf
- 2 phase algorithm: https://kociemba.org/math/twophase.htm