# Cube Tower Puzzle Solution Report

*Antonio Manuel Luque Molina – anluq0157@uit.no*

## *Introduction*

The Cube Tower puzzle is a interesting problem for implement AI algorithms, where you have to organize a tower of colored cubes so that all faces vertically display the same color.

This report documents the approaches taken to solve the puzzle, including the algorithms implemented, the problem instances created, the chosen heuristic for informed searchs, and an analysis of the algorithms' performances. Reflections on the challenges encountered during implementation and their resolutions are also shared.

## *Algorithms Implemented*

### Depth-First Search (DFS)

DFS was implemented to explore all possible configurations of the cube tower by prioritizing depth in the search tree. This approach was helpful in finding solutions that required a smaller number of moves, albeit potentially more time-consuming for deeper solutions due to its exhaustive nature.

### Breadth-First Search (BFS)

BFS was used to systematically explore the configuration space level by level. This algorithm was particularly useful in ensuring that the solution found was one of the shortest possible, in terms of the number of moves required.

### A* Search

The A* search algorithm was implemented with a specific heuristic aimed at minimizing the estimated distance to the goal state. The heuristic counted the number of cubes not matching the most common color in the current configuration, guiding the search towards solutions that progressively aligned more cubes to a single color.

- ➔ **Heuristic Choice**

  For the A* search, a heuristic was chosen based on the number of cubes not matching the most common color in the configuration. This heuristic was informative because it directly correlated with the goal state's proximity, enabling the A* algorithm to prioritize moves that aligned more cubes to a single color.

### Iterative Deepening Depth-First Search (IDDFS)

IDDFS combined the depth-first search's space efficiency and the breadth-first search's optimality and completeness. This approach was effective in finding optimal solutions without consuming excessive memory.

### Bidirectional Search

Bidirectional search was conducted by simultaneously exploring paths from both the initial and goal states until they met. This method significantly reduced the search space and time required to find a solution.

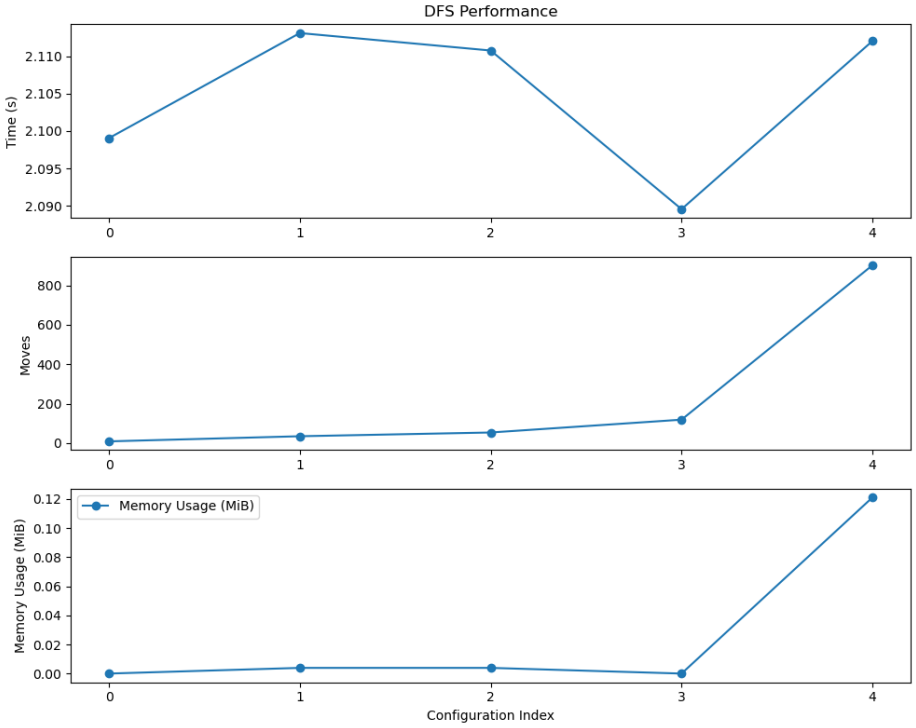
### Problem Instances

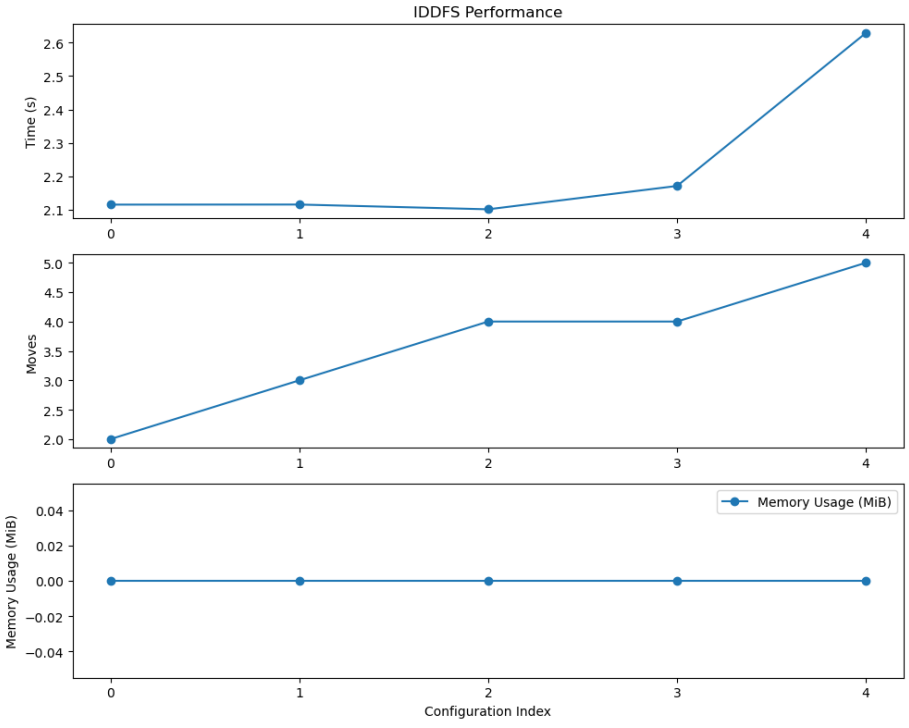
Five problem instances with varying levels of complexity were created to test the algorithms. You can play with them in the code, you can find it in the method initial_configrations().

## Analysis

The performance of each algorithm was evaluated based on time complexity, space complexity, and the number of moves required to solve the puzzle. Time and memory usage were measured empirically, while the theoretical complexities provided insight into each algorithm's expected performance.

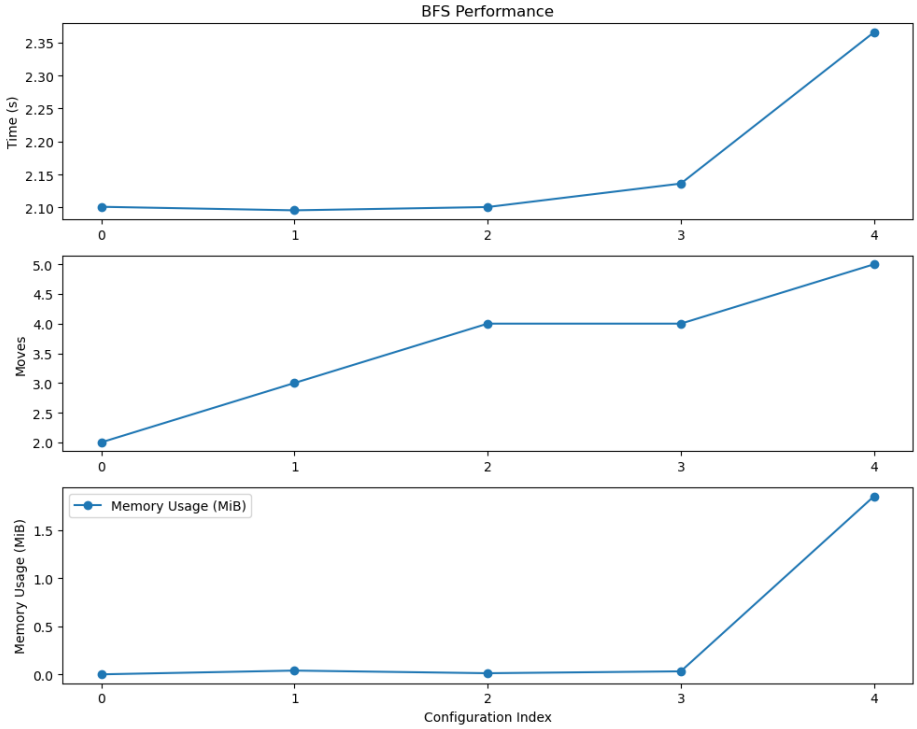
- **DFS** and **IDDFS** were found to be highly effective for shallower solutions but suffered from increased time or moves for more complex configurations.



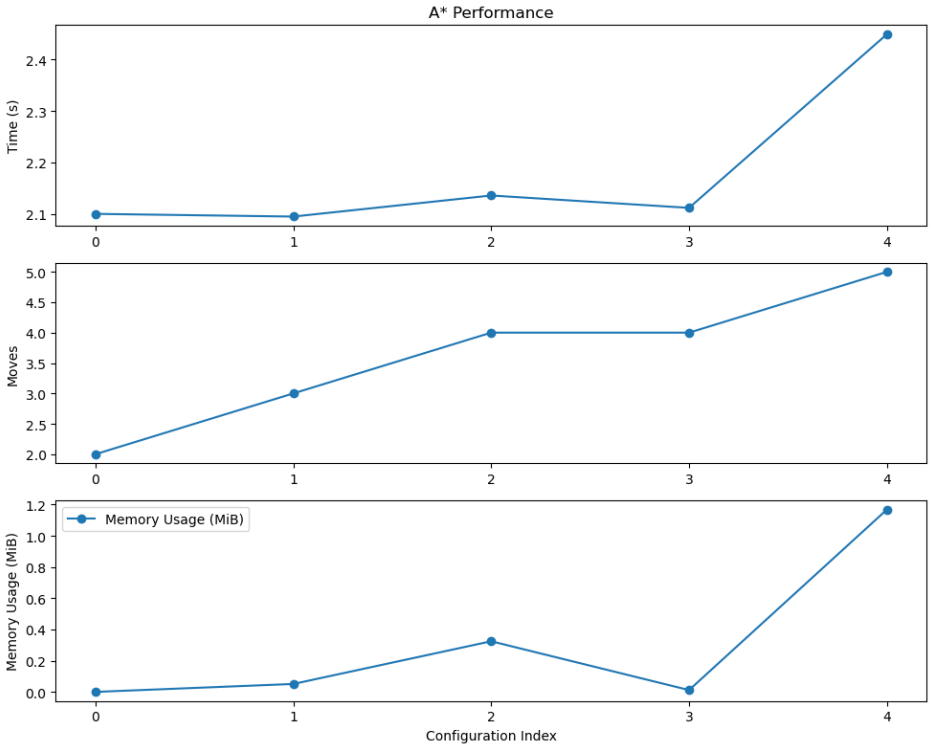




**(All plots and solution paths are in the other attachment, AI Assignment1 Analysis)**

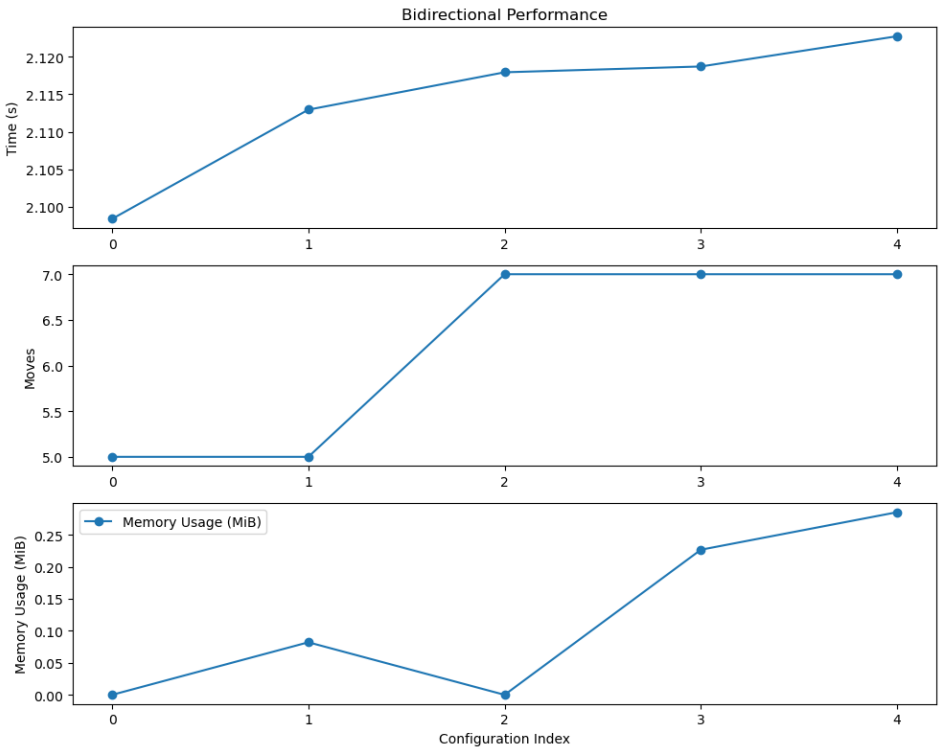
- **BFS** consistently found the optimal solution in terms of moves but was often slower and more memory-intensive than other methods.




- **A***'s performance was balanced, offering a good trade-off between solution optimality and resource usage, thanks to the heuristic guiding the search.




- **Bidirectional Search** showed significant improvements in speed for complex configurations by narrowing the search space.




## ***Reflections on Challenges***

### Challenge 1: Visualizing Solution Path and creating one for the Bidirectional algorithm

Visualizing the solution path was crucial for verifying the correctness of the algorithm implementations.

It was quite complex to implement the method: visualize_bidirectional_path() and to improve the other: visualize_path()

### Challenge 2: Choosing an Effective Heuristic for A*

Selecting an effective heuristic for the A* search was challenging. The initial heuristic attempts did not provide sufficient guidance towards the goal state, leading to suboptimal performance. After several iterations, the heuristic based on the count of non-matching cubes proved to be both effective and efficient.

### Challenge 3: Implementing Bidirectional Search

Implementing bidirectional search presented challenges in efficiently managing and merging the search frontiers from both directions. This was addressed by carefully tracking the paths from the initial and goal states and identifying the meeting point between the two search fronts.

### Challenge 4: Measuring the memory used by each configuration of each algorithm

Measuring the memory in jupiter notebook (also attached in pdf format, called AI Assignment1 Analysis) I had to separate the different algorithms in different tabs and restart the kernel a few times because as you can see in the BFS algorithm, also added in the analysis, the memory came out negative and I guess this is because at certain times the memory is released so it is important to run it in different processes.

## ***Conclusion***

This project explored various algorithmic strategies to solve the Cube Tower puzzle, highlighting the strengths and weaknesses of each approach.

Thanks to this project I have learned the inner workings of several search algorithms used in AI, how to analyze the variations in time, movements and memory between each one of them.