A* on the Megaminx Write-Up

To run code: clang++ -o a_star main.cpp face.cpp piece.cpp dodecahedron.cpp state_node.cpp -Wno-c++11-extensions
(you can also use g++ instead)

For this project, I chose to rewrite my originally unfunctional megaminx code. I learned that the reason I couldn't get it to compile was that g++ was outdate on my computer and once updated, my code couldn't compile for that anyway as there's a number of things Clion will compile but clang++ and g++ won't such as using ">>" when closing a std::map that contains an std::array at the same time.

The rewritten megaminx was written almost entirely by me with the randomizer function being mostly generated by chatgpt. The code also wouldn't be possible without the error checking capabilities of ChatGPT and the Clion debugger.

The completed data structures take the form of:

- A very simple piece class. It knows its unique color set, and has a map from what color it should have to the color of the sticker that is actually in that spot. Edge and corner pieces are distinguished by the size of their set of colors.
    - They have two swap functions, one for corners and one for edges that look complex but are built on the idea that when rotating around a face there is one sticker that stays in place for both, and the others switch places.
    - The sticker can also tell you if a certain color it should have is actually in that spot
- The face class, for which I feel very clever being able to largely automate much more than before. Now instead of the clumsy edge class I realized that if I just have a static map that maps a face's color to five other colors, being its neighbors, and these colors are in a clockwise order, then much of the work can be automated
    - The face knows its neighbors in the form of an array, the pieces on an edge with a neighbor in the form of a map, and it's color.
    - It has two helper functions to set up its array and map, allowing me to create the face object and finish it later
    - They can rotate clockwise and counter clockwise, with the idea that if you want to rotate clockwise you just swap two corner pieces (0 and 1), move to the next in the counter clockwise direction, swap the new one and the one next to it (1 and 2), and so on swapping them four times you will have simulated swapping the pieces in a counter clockwise motion. For some reason edge pieces should be swapped clockwise and I have no clue why I just took that at face value when it worked. Reverse the directions you do the swapping for counterclockwise and that's how each works
- The dodecahedron class, which is long but essentially all it does is print and set up the face and pieces correctly. It does nearly no work outside of that.

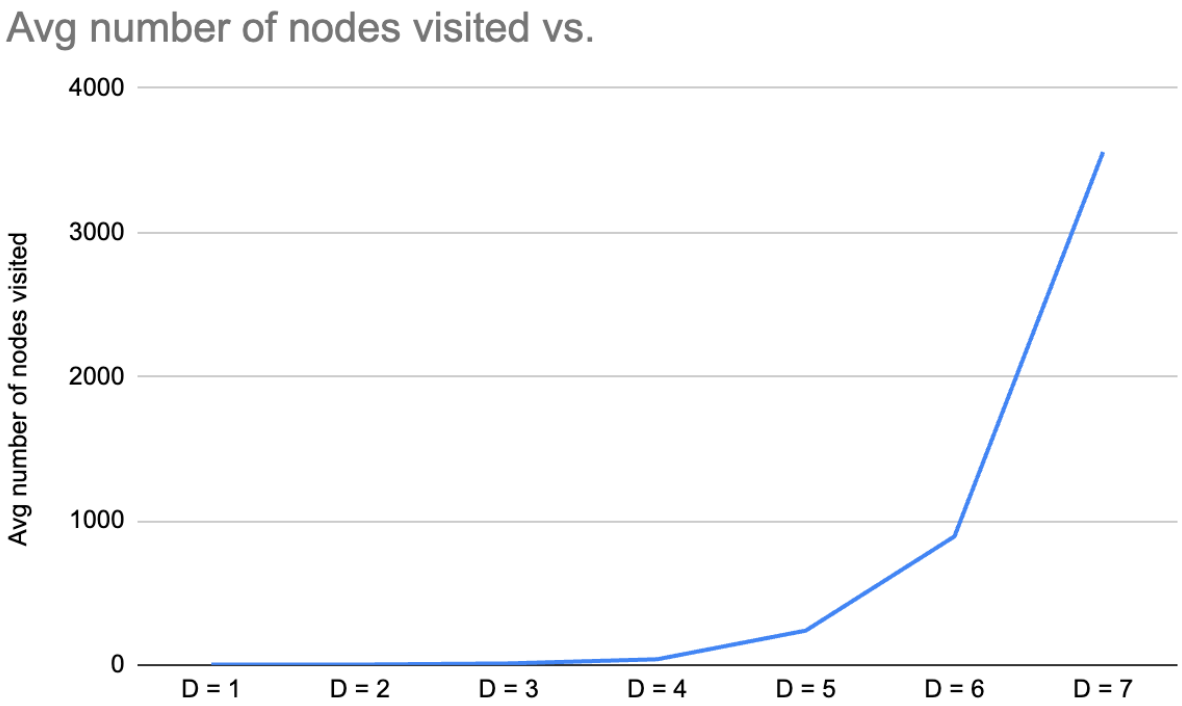
    - One thing that I'm *super proud of* is a working copy constructor which is amazing. It saved me so much time on the whole making child nodes thing.
- A state node class, which contains a dodecahedron, the knowledge of if it's solved, the depth in the tree, and the cost it took to get here which is the parents cost plus the heuristic.
    - It also has a working '>' operator which saves a lot of time on the A* part, the ability to print, and the ability to make 12 child nodes based on it (and some functions to access it's private data members)

The heuristic that I used had one very important feature, and that was how easy it was to implement haha. So when discussing with my coding buddies we realized that if you consider a rotation relative to a face, one rotation changes one edge piece and two corner piece. So if you have two edge pieces out of place it requires two rotations to fix. The number of corner pieces out of place will give a heuristic value of ceiling(num_corner_out_of_place / 2). Take the higher between these two for a face, take the higher of these two for a face, and take the highest of all the faces and that's your heuristic.

The way my A* code works is simple, which is a nice change of pace.

- Make a priority queue
- Put in a scrambled state
- Start while loop: check if we're solved
    - Make the current state the top of the queue
    - Pop the queue
    - Make a bunch of child nodes based on current node
    - Put them in the queue
    - Print the current state

Thankfully the state node's '>' operator and the dodecahedron's copy constructor do almost all the heavy lifting for us



I learned a lot from this exercise, mainly this solidified what AI was to me in my mind. AI has always been a nebulous concept, some black magic that makes intelligent decisions. But from this I learned that a humble pathfinding algorithm can be used in AI. If you told me about A* outside of this class I wouldn't have thought to implement for anything other than pathfinding, but now I see that AI is not always as mystical or complex as I thought and the simpler algorithms I learn can be used to solve complex problems.