A\* on the Megaminx

9/20/23

**How to run:**

The program consists of two files, *megaminx.py* and *astar.py*. The former handles the GUI and data structure representing the Megaminx and the later solves the toy using the A\* algorithm.

The program is called by running *megaminx.py*. I ran the program using [Anaconda](https://www.anaconda.com/) with Python 3.11.5 installed.

* python <parent\_dir>/megaminx.py

The following packages are required:

* Tkinter Version 8.6
* math
* random
* copy
* time

All of the required functions can be run from within the Tkinter GUI shown below. The GUI is intended to be run in full screen.

A screenshot of a computer

Description automatically generated

Select a face to rotate by clicking the yellow square on the top right of the screen and choosing the color that corresponds to the center of your desired face.

Press the “Rotate Clockwise” or “Rotate Counterclockwise” buttons to perform the operation on the face that is currently selected.

To scramble the cube, enter an integer value into the text box under “# Random Rotations” and press scramble. In this version of the program, scramble will only perform clockwise rotations.

Press the “Solve using A\*” button to solve the Megaminx. Text will be printed to the terminal to indicate the start and completion of the A\* algorithm, as well as the number of nodes generated and the required steps to a solution. The GUI will show each turn back to the solution with a slight delay in between.

The solver can be verified by rotating any face of the Megaminx counterclockwise. The solver only implements counterclockwise rotations and will make four additional moves to restore the face.

**Data Structures used in representation of the Megaminx:**

The Megaminx is represented by 12 face objects and 120 piece objects.

Pieces are the same thing as “stickers” on the toy. Each piece contains two sets of coordinates, one holds the starting position of the piece and the other holds the current position as the piece is “moved” around the dodecahedron.

Coordinates are written as: [face (0-11), position on face (0-9)].

The main data structure of the program is a single array containing a pointer to each of the 120 piece objects. I will refer to this array as the “logical array”. The index of a piece in the logical array corresponds to its coordinates.

Each face object contains four smaller arrays for the internal corners, internal edges, external corners, and external edges assigned to that face. Corners are the stickers that have six degrees of available movement and edges are the stickers that have four degrees of available movement. Internal refers to stickers that are adjacent to the center of the parent face, while external refers to stickers adjacent to the center of another face.

The four arrays contain indices to the logical array that correspond to the corners and edges that belong to this face. When a rotation is performed on a face, the current coordinates of the piece objects represented by the indices are swapped leftward or rightward in each array for a clockwise or counterclockwise rotation.

**Data Structures used in A\*:**

The A\* algorithm is implemented in *astar.py* which is called from the GUI. It is passed the logical array data structure that represents the Megaminx. The Megaminx structure is stored in a node object that is then placed on the priority queue “open\_list”. As more nodes are expanded, they form a search tree with 12^h nodes on each level. When A\* is completed, nodes are pushed onto a stack while traversing to the root node which are later popped off to show the solution. Both the priority queue and the stack are implemented using python lists

**Randomizer:**

The randomizer is implemented in the function do\_scramble(). It takes an integer number for the number random moves that are desired to be made. For each move, a random face and a random direction are determined, and the rotation is applied using the data structures described above. A check is in place to prohibit the same face from rotating and then immediately un-rotating. The code submitted with this assignment always makes clockwise rotations to simplify solving.

**Heuristic:**

I decided to implement a superior heuristic described in the week 4 class discussion post.

Rotating any face will affect the positions of 35 stickers. 10 of these stickers are adjacent to the core of the rotating face, while 15 of the stickers are adjacent to the core of a different face.

I count the number of stickers that are currently on a face other than their home face. Stickers that are on their home face but in the wrong spot are not counted. Therefore, at most 15 non-core-adjacent stickers are counted for each face. Since the core-adjacent stickers are not counted, I avoid double counting stickers that are non-core-adjacent for a different face.

For exactly one face rotation to solve the Megaminx, all 15 non-adjacent stickers of that must be on a face other than their home face. If I take the floor of the sum of stickers not on their home face divided by 15, I exclude all faces that require either zero or more than one rotation to approach a solve. This lowers the potential maximum of my heuristic to 8, but it ensures that the heuristic is admissible because faces that take fewer than one rotation to be solved are never counted.

**Plot of node expansion:**

**What I learned:**

This project taught me about the consideration one must have when writing a program that runs in exponential time. While I have always tried to follow best practices when programming, I’ve always been much more concerned with demand on the processor than demand on memory. This was reasonable because up until now I had never written anything that required upwards of 16gb of memory to run.

Many of my peers used the same heuristic as me but were able to solve larger numbers of scrambles because their data structure representing the Megaminx took up less memory. I chose a memory intensive data structure in the first project because it let me automatically build and rotate the Megaminx without having to hard code several relationships.

If I had more time to work on this project, I would rethink my data structure to use the best of both worlds. I could have a single array of integers, where each integer corresponds to one cubie on the cube. A solved Megaminx would have the index of the array match the integer value of the cubie. Every face would have a map of cubies they affect and perform a rotation by moving integers within the array. This would drastically reduce memory usage, but it would increase processor burden because I would have to convert between the cubie representation and the sticker representation that my GUI uses.