Ryan Gaines

Dr. Goldsmith

CS 463G: Intro to Artificial Intelligence

4 September 2018

Program Assignment 1: Megaminx

In this assignment, we were tasked with creating the twelve sided Megaminx, a puzzle based cube similar in nature to a Rubik cube. The following will describe how the data structure for the Megaminx is represented, how the randomizer works, and an admissible heuristic for estimating the solution to the Megaminx. My version of the Megaminx was implemented in python. To execute, type *python main.py* in the directory containing the files.

I based my implementation off a model found from Ruwix website. For each of the sides, it was easy to see that moving one piece affects all of the neighbors, and caused pieces from neighboring side to be transferred to another neighboring side. I decided to number the sides of the cube 1-12, and develop my implementation as a list of objects that represent sides. Each side object would have an array that contains 10 characters that represent the movable pieces, and one character that represent the center piece. It would map the movable side pieces as such, where 1-10 could switch and interchange between sides, and the center is always stagnant:

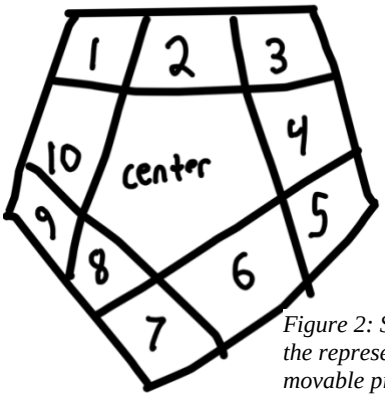


Figure 2: Showing the representation of movable pieces in array

Then, I analyzed a physical Megaminx and noticed which sides neighbored which. The image provided from the Ruwix website was instrumental in helping me figure out which sides are bordered with the other sides. Once I discovered how each side was related to the others, I hard coded in their relations, giving each object another array of

Figure 1: Image from Rubix's Online Megaminx

neighboring sides. Each neighboring side is numbered 1-5, and rotate around the pentagon counter clockwise, resulting in the sides numbered as such:

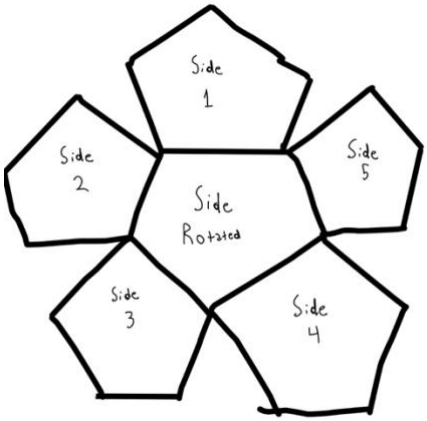


Figure 3: Relationship between side spun and neighboring sides

Once all these relationships were hardcoded for all 12 sides, it then began time to implement how the switching between the sides would work. I found that when rotated clockwise, each time 3 from side 1 went to side 5, and 3 from side 5 went to side 4 and so on. I determined using Figure 4 which 3 went to the new side, based on off how the side class is structured for clockwise motion. I also coded the counterclockwise motion due to forgetting the direction of clockwise at first implementation. I also used the rotate function to rotate the called upon the switching of the positions in the array of the rotated face to match up with the representation of the side object as described earlier.

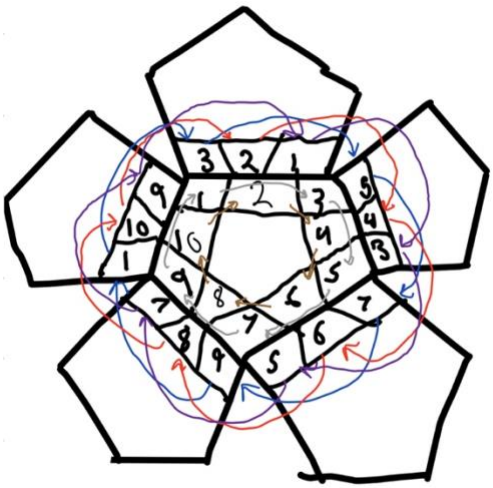
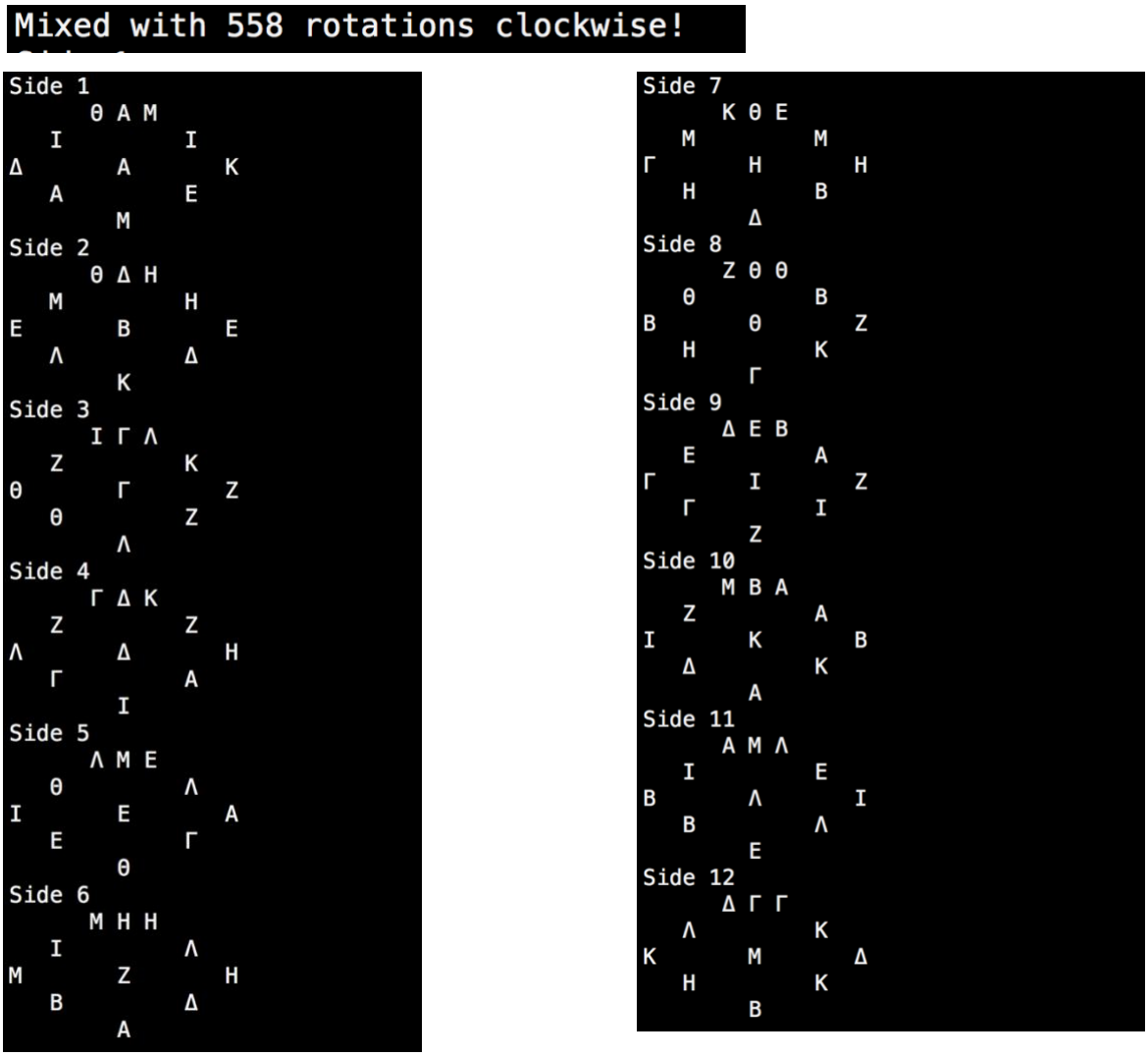


Figure 4: Showing how the switching between sides works out.

My version of the code allows you to manually mix up the cube, as well as randomize the Megaminx using a random functions. First, it chooses a random number between 2 and 1000 for the number of rotations that should be done. Two was chosen due to a ruling found on the World Cube Association competition guidelines stating that in competitions it must be at least two moves from solved. From that, it choses a side randomly, and then rotates that side clockwise.

All the random functions are seeded based on system time, which allows for different combinations when ran consecutively. The downside of this is that since we are using seeding from system time, it becomes predictable the outcome of the random, since we are able to access the random based on system time and easily replicate what the rotations will be. To execute the randomizer, type *python main.py*, and when prompted type *r* to randomize the Megaminx.

Running the code resulted in this when ran at a system time of 2018-09-04 23:01:29.852335 resulted in the following for the GUI output, where side 1-12 is based on Figure 1.



An admissible heuristic for this problem would to count the number of outer wall movable cubes that do not match the center value, and then divide that number by 15. If the number given is a decimal, then use a floor function to round. This is an example of the Hamming Distance, which is counting the number of misplaced parts. The Hamming Distance is a method frequently used to calculate admissible heuristics in fifteen puzzles. We would divide by 15 to factor in how in each move, 15 pieces are moved to a new side. It is admissible because in a best case scenario, the number of moves to fix the Megaminx is the same as number of parts off and re-put on. In real case scenario, the number would most likely not be the same as this in most cases, meaning that it is indeed a lower bound. To implement in the code, one would just run and see how many of the cubes sides are the same as the center, and then divide that number by 15 and take the floor function of that.

Using this method in the version above gives an admissible heuristic of 7. This is because in total, there are 105 pieces that do not match the center unmovable piece, and when dividing by 15, we then get 7. This is admissible for sure because it will take more than 7 moves, given it took over 500 to get to the configuration, meaning that it is indeed a lower bound estimation.

In this project, I learned how to code a complex real word item into code using data structures. I also learned how to divide tasks into subtask better. This project also got me more familiar with how admissible heuristics work, and how it is a lower bound estimate for determining the better path in search trees. This project has gotten me to began thinking how this heuristic will play a part in the next assignment, and how it could be used to solve the cube by preferring the path of lower heuristic.

Works Cited

Online Megaminx Simulator, < https://ruwix.com/online-puzzle-simulators/megaminx-simulator.php>, Published 2017, Accessed September 4, 2018

Admissible Heuristic, < https://en.wikipedia.org/wiki/Admissible_heuristic#cite_ref-1>, Last Edited March 11, 2018, Accessed September 4, 2018

WCA Regulations, < https://www.worldcubeassociation.org/regulations/#article-4-scrambling>, Published January 1, 2018, Accessed September 4, 2018