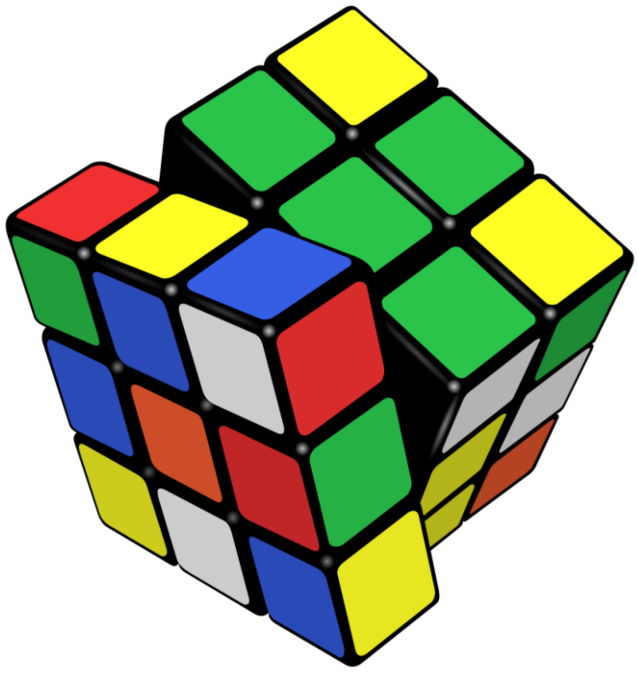
P2: Problem Formulation in Python

Rubik's Cube through SOLUZION



Caelen Wang & (Brian) JoonHo Lee

August 31, 2017

Word count: 940

(1) What is the problem you have formulated? If this is not a well-known problem, explain what the problem is and any rules you have made up.

The problem we have formulated is the Rubik's cube puzzle. The Rubik's cube is a three by three cube with nine tiles on each face, 54 tiles in total. There are nine tiles for each of the six colors: white, yellow, red, orange, green, and blue. The cube's three horizontal layers and three vertical layers rotate 360 degrees independently. The problem/task presented by the Rubik's cube is for the player to arrange the cube in such a way that each face has a distinct color by rotating the layers. The problem is especially challenging because the player needs to have a systematic approach to solving the cube. Simple trial and error will not lead to the solution since the Rubik's cube has a total of 43,252,003,274,489,856,000 possible arrangements. A brute-force method will lead to a combinatorial explosion.

(2) What is your state representation and why?

Our state representation consists of six distinct three by three matrices representing the six faces of the Rubik's cube. The state representation is stored in a list of six numpy two-dimensional arrays. This is a simple yet accurate representation that is easy to manipulate. The textual display has four matrices in a row: left, front, right, back, respectively. A matrix above the front matrix represents the top face, and a matrix below the front matrix represents the bottom face. We believe that this display is intuitive for the player since he/she can visualize the state of the cube.

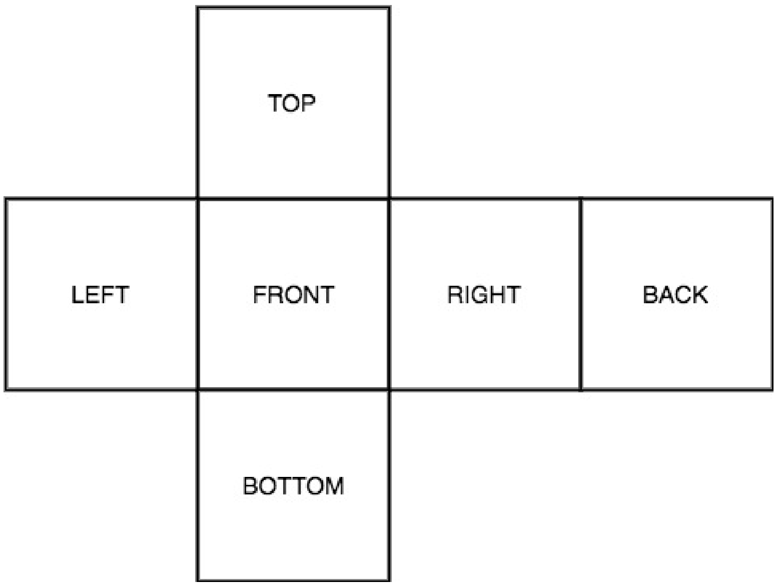


Figure 1: State representation of the rubik’s cube. The 6 sides laid out in a floor plan-like drawing provides an intuitive visualization of the problem.

(3) What operators do you provide? Did you face a choice when you designed your operators, and if so, what is another possibility that you considered? Why did you go with the choice you made?

The set Gamma contains 10 distinct operators: $\Phi = \{\varphi_0 \ldots \varphi_9\}$ The player can rotate the top and bottom layers left or right (2 layers 2 rotations = 4 operators), the left and right columns up or down (2 columns 2 rotations = 4 operators), and rotate the entire cube left or right (2 operators). We believe this set of operators to be the most intuitive for the player because it strikes a balance between simplicity and functionality. Originally, we had 18 operators – additional ones for turning the middle layer and column, as well as the left-right direction without rotating the entire cube. We realized that turning the middle layer/column is unnecessary since it’s equivalent to turning the two other layers/columns. Additionally, by simply adding two operators for turning the cube, the column rotation operators are suffice; therefore requiring only 10 operators.

(4) Did you provide a visualization in your formulation? Briefly, how does that work?

We provided state visualization using the Tkinter module. Given that we follow the same representation as figure 1, we created a 9x12 numpy array that would have each of its 'tile' colored accordingly. Any tile that is not part of the cube will automatically be colored black, while the other tiles' colors are replaced depending on their elements' values. e.g. 0 for red, 1 for green, 2 for orange.etc. To effectively compute this process we first initialize the array with every tile as black, and render each tile with a for loop. Therefore whenever a move is made by the player the program would iterate through each tile and re-render the visualization.

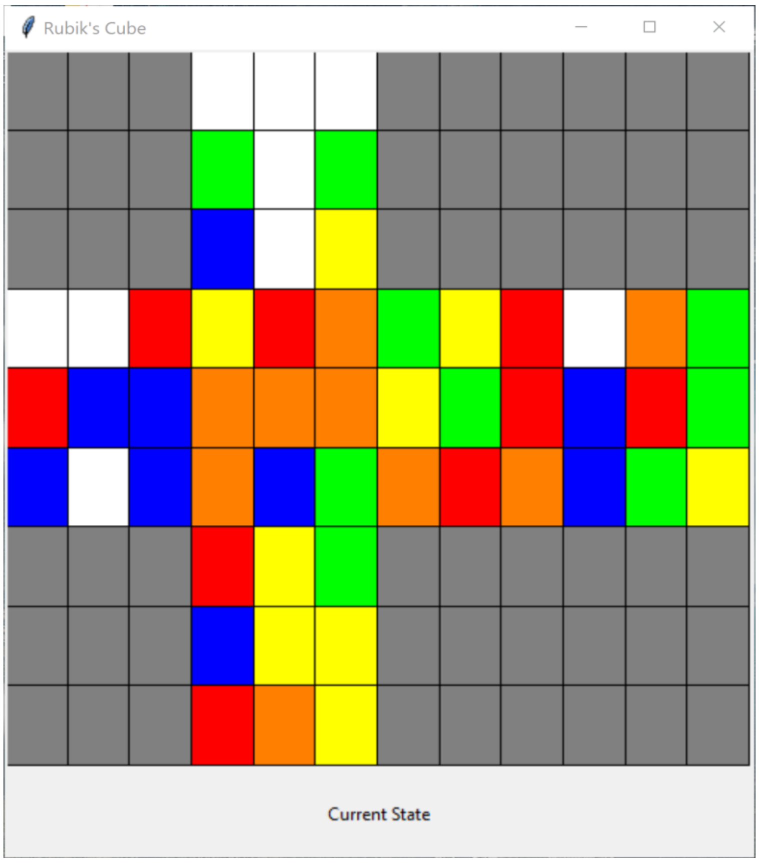


Figure 2: exemplar visual representation of current state. This was built using the Tkinter module

(5) Describe how you and your partner divided up the work of the project.

Since the formulation of the problem can be broken down into different components, the entire process was divided into multiple tasks that were then distributed among the two members of the group. Once every module of the code was completed, tested, and debugged, they were then put together into large .py files for final testing. The final testing took a while to process as bugs would often appear during the assembly of the final product. The tasks were mainly divided into two parts: the game and the visualization. Components of the game algorithm were prioritised. Once the codes designed for the game were completed and tested, the visualization process was completed and combined with the game algorithm.

(6) Describe any particular challenges you had with this project.

Coming up with a concise yet practical state representation presented a challenge. The 6 arrays are then concatenated with 6 other arrays that holds a non-related value (such as 999, which has no relation to the color being used for the cube) in order to output a 9 by 12 array as a final state representation, which would most resemble state presentation illustrated in figure 1. A more specific problem occurred during writing of the *move()* function, in which we found difficulty in exchanging values of elements of arrays representing different sides of the cube. Most importantly, the greatest challenge came after the completion of the ‘game’ component of the project. Formulating state visualization for rubik’s cube through SOLUZION and Tkinter proved to be the most difficult task, with problems arising from data type errors, window not popping up upon running the program, to the window only showing one column of the entire state presentation, which is a 9 by 12 2-dimensional numpy array.