Michael Liao
CSC 372

A1 Lab Report - Modeling a 2x2x2 Puzzle Cube in Python

Representation as a Data Structure

I modeled the virtual cube after my own cube, which had a white-yellow **red-purple** blue-green color scheme (unusual coloring emphasized). Every 2x2x2 cube consists of 8 cubelets, which I represented with its own **Block** class and a flat array named **blocks** within the **Cube** class. As seen in **Fig 1**, each cubelet had a distinct triplet of coordinates, ranging from (0,0,0) to (1,1,1) in the XYZ plane. These coordinates map to binary representations of the numbers 0-7, perfect for use with **blocks** as indices.

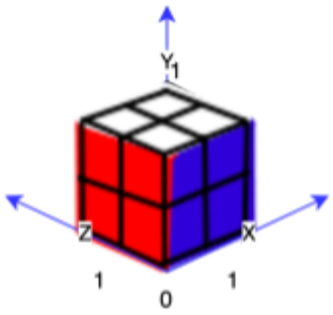

Fig. 1: Sketch of the 3D plane in which the cube is oriented.

The Block Class

Each cubelet tracks its **neighbors** in each of the directions (X, Y, and Z), as well as its **coords** and **faces** (X, Y, and Z, understood to be the face *along* the relevant axis) in lists. The **faces** can be considered to be the stickers on the cubelet: this way, I can account for changing rotation of the cubelet itself as well as full rotations of the cube's faces. Six characters map to the different colors on the faces of the cube:

1. R → Red
2. W → White
3. B → Blue
4. P → Purple
5. Y → Yellow
6. G → Green

These are used to uniquely identify each cubelet in **Cube** as well as print out a net of the cube in **display()**. To situate the cube, I designated the blue face as the front face (all others can

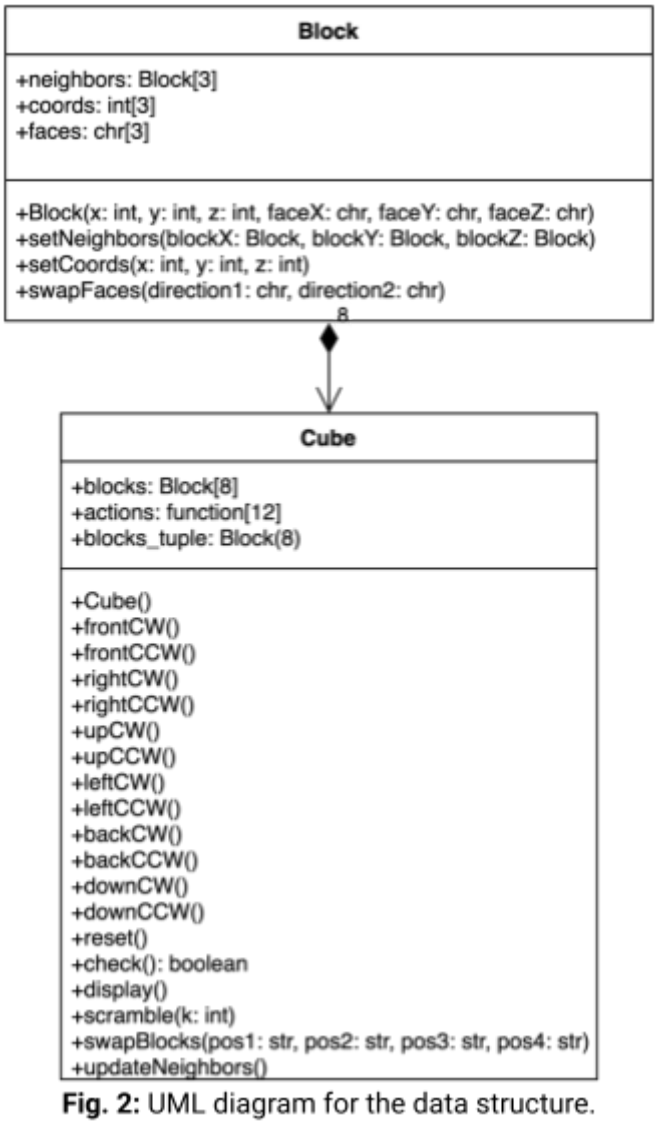

Fig. 2: UML diagram for the data structure.

be determined relative to it by unfolding the cube).

The class contains setters for these attributes. Of note is **swapFaces()**, which only ever swaps two faces at a time (since each rotation move can only move a piece along one plane, preserving the orientation of one of the faces).

An illustrative example: Take the Red-White-Blue block (RWB) in its initial state.

- The red face is along the x-axis (facing the negative x direction),
- the white face is along the y-axis (facing the positive y direction), and
- the blue face is along the z-axis (facing the negative z direction).

Let us perform the F rotation on it (see **Fig. 3** for an explanation of standard puzzle cube notation). In such a move, the blue face's four pieces would stay together, but the corners would have changed spots. We can then fix the blue face and swap the red and white faces, as the block now has:

- the red face along the y-axis (facing the positive y direction),
- the white face along the x-axis (facing the positive x direction), and
- the blue face along the z-axis (facing the negative z direction).

For my purposes, the actual sign of the directions of the faces do not matter; the directions are tracked to enable the 3 possible orientations of each piece.

Any character followed by an apostrophe (') is read as *character prime*. This indicates a counter-clockwise quarter turn, as opposed to the default clockwise quarter turn. (For example, F' would undo F.)

A character followed by the number 2 indicates to rotate that face twice.

- F → Front face
- B → Back face
- R → Right face
- L → Left face
- U → Upper face
- D → "Down" face

Fig. 3: Standard cubing notation.

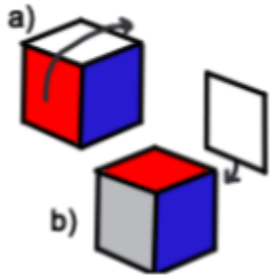

Fig. 4: a) RWB, initial state.
b) RWB, post-F rotation.

The Cube Class

The cube is initialized in a solved state, (recall: with blue as the front face). This is done by instantiating all the cubelets and storing them in the relevant **blocks** and **blocks_tuple** (used to maintain a known order, since the list **blocks** will be modified to reflect changes to the cube, whereas **blocks_tuple** will remain static).

The bulk of the functions in **Cube** handle rotations of the different faces (**frontCW()**, for example, rotates the front face clockwise). These make use of the helper functions **swapBlocks()**, which swaps the four affected cubelets around within **blocks**, and **updateNeighbors()**, which iterates through **blocks** and ensures all cubelets' new neighbors are accounted for after a rotation.

The static **blocks_tuple** is utilized in **check()**, where I take advantage of the fact that in a 2x2x2 puzzle cube, if two opposite corners (I chose the pieces initially in (0,0,0) and (1,1,1), so RYB and PWG) have all three neighbors they are supposed to have when solved, the entire cube must be solved (thus eliminating the need to establish an absolute frame of reference for checking the cube and then full rotations of the cube into other planes that are still solved).

If such a layout exists, it indicates one of the cubelets has been assembled in an incorrect orientation (and therefore the cube cannot be solved). The proof for this exists, I am sure, but I do not possess the skill just yet to write it myself; it must have something to do with the net orientation change achieved by a rotation of a face being different from the net orientation change in a flipped corner.

Fig. 5: An impossible state of the cube (that has two opposite corners with proper neighbors, but is unsolved).

The final two functions are **scramble(k)**, which perform **k** random moves on the cube (taking care not to undo previous moves) by using the **actions** list; and **display()**, which prints out a net of the cube as shown in **Fig. 7** further on. **Fig. 6** is a rough diagram of the cube's initial net when it is instantiated.

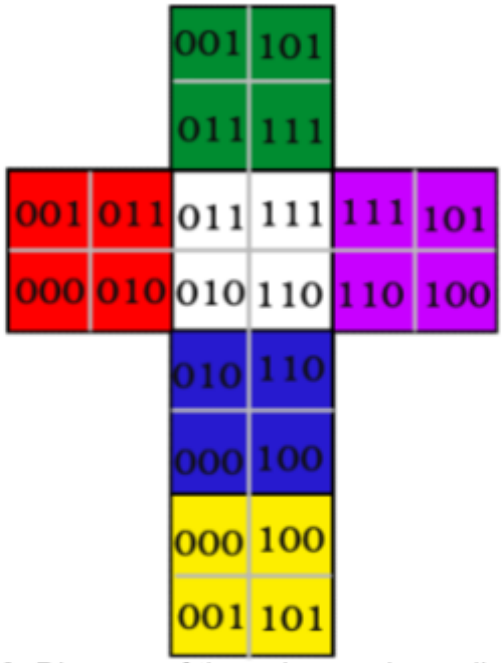

Fig. 6: Diagram of the colors and coordinates associated with each part of the net (condensed into a 3-digit binary string).

Usage

In the **main()** function, a switch statement checks for standard cube notation inputs and performs the rotations, displaying the cube afterwards, on repeat (until solved).

The program starts out by scrambling the cube with 10 actions, then prompts the user to enter moves to attempt to solve it. In **Fig. 7**, I demonstrate the program after I enter a single move (F2), and the cube net updates accordingly.

```
Use standard cube notation to rotate cube clockwise (F, B, R, L, U, D).
Add a ' symbol (read: prime) to indicate counter-clockwise, and 2 to rotate twice.
Ex: F rotates the front face clockwise. F' rotates it counter-clockwise. F2 rotates it twice.
------
    G G
    G G
R R W W P P
R R W W P P
    B B
    B B
    Y Y
    Y Y
-----------
Scrambling!
    P Y
    R B
W W G R Y R
R Y B Y P P
    P G
    W W
    B B
    G G
-----------
-1 to exit.
> F2
Rotating F2.
Still unsolved...
    P Y
    R B
W W G R Y R
P P B B R Y
    W W
    G P
    Y B
    G G
-----------
-1 to exit.
>
```

Fig. 7: Terminal output from running **solver.py**.

Heuristic Proposal

There are a variety of options: Hamming distance, for example, involves counting stickers. Straight-line distance is guaranteed to be admissible, but I find it to be less efficient for my needs, since the maximum straight-line distance on this coordinate plane would be $\sqrt{3} = 1.7$. A potentially more efficient heuristic would be Manhattan distance, though proving it is harder; however, I would prefer to have a more effective heuristic to minimize the time I must spend searching.

In this case, Manhattan distance would be calculated by setting an absolute frame of reference by arbitrarily selecting one cubelet as being in place (it may be more

computationally expensive than it is worth to determine which cubelet has the most "proper" neighbors and set the frame of reference according to that cubelet, since in my implementation I have to iterate through every cubelet and compare "proper" neighbor counts) and determining how far away every other cubelet is from its proper location, then summing that.

To account for the fact that each rotation moves four cubelets, we then divide that sum by 4. Otherwise, a cube that is one turn away from its solved state would have a heuristic score of 4, when in reality it was one turn away (making it inadmissible!).

To recap: $h(n)$ = sum(# cubelets away from solved, for each cubelet)$/4$.

I believe this would be admissible. Dividing by four ensures that our heuristic never overestimates the amount of turns required to solve the cube; it can only match it, like in the earlier case. If we have a total of 5 cubelets out of place, it will take *at least* two turns to move them all back into place (assuming best case), and $5/4 = 1.25$.

Takeaways

In this assignment, I learned to break a problem down into its component pieces and focus on building a structure that can maintain itself. For example: I knew I would have to check for state symmetry (and ensure that a solved cube was considered solved no matter its rotation), so I built that into the data structure by having each cubelet keep track of adjacent cubelets. Instead of storing the 3 million or so different states, I created a cubelet that has three different orientations of its faces and then accounted for the positions in the cube in the cube class. This way, I would not have to check three different states; just two corners, as described earlier.

Acknowledgements

- Dr. Allen, for granting me an extension on this assignment.
- Lam and Chantakrak for being astounding moral support and having a late-night heuristic think session. (And for some snacks!)
- Pierce, for keeping me on track when I got distracted in the dorm room.