CSC 440 Assignment 3: Rubik's Cube Solver

Out: Thursday, February 23

Due: Monday, March 6, by 11:59PM

Introduction

This is a solo assignment

In this assignment, you will develop an algorithms for solving the $2 \times 2 \times 2$ Rubik's Cube, also known as the Pocket Cube. Call a configuration of the cube "k steps from the solved position" if it can reach the solved configuration in exactly k twists, but cannot reach the solved configuration in any fewer twists. The rubik directory in the problem set package contains the Rubik's Cube library, as well as a graphical user interface to visualize your algorithm. We will solve the Rubik's Cube puzzle by finding the shortest path between two configurations (the start and end) using BFS. A BFS that goes as deep as 14 levels will require a great deal of memory. However, the number of nodes at depth 7 is much smaller than half the total number of nodes. So, we can do a two-way BFS, one starting from the starting configuration and the other from the solution, and try to meet in the middle.

You will be provided with a zip file on TopHat, `rubik.zip`, which contains a unit-test framework, a library that gives you a representation of a Rubik's cube, and even a tool for visualizing a cube. However, this visualization tool may not work properly in all Python versions, but you do not need the visualization tool to complete the assignment.

You must implement the function called `shortest_path(start, end)` in `solver.py`. This function takes two positions, and returns a list of moves that is a shortest path between the two positions.

Extra credit (10%)

Prove that for any starting configuration C, and any sequence of moves S of any length, if S is repeated enough times, you will eventually return to C.

If you do the extra credit, you may either upload it as a PDF or a text file, and (just so we don't miss it) mention in the comments at the top of `solver.py` that you have done the extra credit.

For this assignment, your solution must be in Python.

We will evaluate your solutions in Python 3.7.

Solo Assignment

This is a SOLO assignment. You may not share your code with anyone in the class or look at anyone's code. Whiteboard discussions with diagrams are perfectly acceptable.

Lateness

Submissions will not be accepted after the due date, except with the intervention of the Dean's Office in the case of serious medical, family, or other personal crises.

Grading Rubric

Your grade will be based on three components:

- Functional Correctness (25%)
- Design and Representation (50%)
- Invariant Documentation (25%)
- Extra credit (10%)

Design and Representation is our evaluation of the structure of your program, the choice of representations (how do you store the moves? how do you efficiently search a graph?), and the use of appropriate (not excessive) comments. We have given you some obvious hints in the file `rubik.py`

Invariant Documentation is to force you to reason about the running time of the algorithm, as well as its correctness. Whenever you have a loop in the body of your algorithm, you should state the invariants that hold for that loop.

- Initialization: how is the problem set up?
- Maintenance: how do I know I'm making progress? how is the problem getting smaller with each iteration?
- Termination: how do I know I'm done>
- Usually, these should all be closely related.

Hints

Your solution should be callable as `shortest_path(start, end)` in `solver.py`. It should take two positions, and return a list of moves that result in transforming

a $2 \times 2 \times 2$ Rubik's cube from the given `start` position to the given `end` position.

There is a lot of complexity to the representation in `rubik.py` but you don't need to understand the representation of configurations as permutations of cubies. You are welcome to dive deeper into, it, but here is what you need to know:

- There is `rubik.quarter_twists = (F, Fi, L, Li, U, Ui)` which is a tuple of all possible quarter-twist moves. Their names are mnemonics: Front, Front-inverse, Left, Left-inverse, Upper, Upper-inverse. From a perspective of holding the bottom-right-rear cubie fixed in space, these are all the possible moves.
- The function `rubik.perm_apply(perm, position)` takes a move (permutation) and a position (configuration of the cube), applies the move to the position, and returns the resulting position.
- The function `rubik.perm_inverse(p)` takes a move and returns the inverse of that move. For example, `rubik.perm_inverse(F)` returns `Fi`.
- A permutation is just a tuple of cubie faces in a particular order, so tuple equality can be used to compare two positions.
- Given all of this, you should be able to *generate* a graph of the configuration space and solve the problem.