# cs221 Project Progress Report

Cristian Lara – claral3@stanford.edu – 5973257

November 16, 2017

---

# Rubik's Cube Solver

- ## Model: Search Problem

**State:**
Let a state be a Rubik's Cube object which keeps track of its 6 faces. Specifically, the Cube keeps track of a map from 6 face identifiers ['L', 'R', 'F', 'B', 'U', 'D'] to 6 arrays of length 9 containing the colors on the 9 positions on the face (represented as an enumeration of the 6 colors). For example, an unsolved face with the top row being color 3 and the rest being color 1

| | | |
|---|---|---|
| 3 | 3 | 3 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

would be represented as an array [3, 3, 3, 1, 1, 1, 1, 1, 1]. If this were the front face of the cube, it would be represented in the dictionary as {'F': [3, 3, 3, 1, 1, 1, 1, 1, 1]}. But this is only a single face; a solved cube would be represented as so:

```
{
    'U': [0, 0, 0, 0, 0, 0, 0, 0, 0],
    'L': [1, 1, 1, 1, 1, 1, 1, 1, 1],
    'F': [2, 2, 2, 2, 2, 2, 2, 2, 2],
    'R': [3, 3, 3, 3, 3, 3, 3, 3, 3],
    'B': [4, 4, 4, 4, 4, 4, 4, 4, 4],
    'D': [5, 5, 5, 5, 5, 5, 5, 5, 5],
}
```

**Start State:**
A Rubik's Cube in any position: solved or shuffled.

**End State:**
A Rubik's Cube in a solved position (i.e. the map shown above representing the solved cube).

**Actions:**
For every state, there are exactly 12 actions which can be performed on it, 2 actions for

each face. Each face can either be rotated clockwise or counterclockwise. Implementation-wise it's important to note that rotating the values in the array representing a face is not enough, you must also properly swap the values of the faces adjacent to the rotated face.

**Successors:**
The successor states resulting from each action are simply the Rubik's Cube objects with the updated/rotated faces.

**Cost:**
*Explained in Algorithm section*

- # Algorithm: Uniform Cost Search + A*

  **USC:** For the basic Uniform Cost Search algorithm, the cost is just how many turns it takes to find the solution (i.e. cost of 1 for every turn). This cost function equally prioritizes all the actions with the fewest ancestor actions. It is guaranteed to find a solution with the fewest number of actions, but can naively blow up the search space since moves which take us in the right direction are not prioritized over moves taking us in the wrong direction. This was explained in lecture as expanding the frontier of UCS equally in every direction including the wrong direction.
  In this specific case, simple UCS first performs all the 12 rotations on the start state, checks if any of those give us the solution, then expands the search space to include every move that is 1 move further than the last one. This is wasteful because most of the possible moves actually scramble the cube further.

  **A*:** To improve the search algorithm, we implement A* and modify the cost function to incorporate not only the distance from the start state, but also the distance from the end state. To do this, we come up with a consistent heuristic which assigns higher costs to actions that further scramble the cube. Conceptually, we are punishing increased entropy in the Cube's faces.

  Specifically, it is generally true that a cube with fewer colors swapped around is closer to being solved than a cube with many colors swapped. We can incorporate this into the cost by summing the entropy of each face after an action. What I mean by this is to loop over every face and count how many distinct colors are on each one. For a fully solved cube, the total cost is just 6 because there is one distinct color on each face. But if we were to rotate the top face, the top and bottom faces would still have costs of 1 each, but each of the other faces would have 2 distinct colors on them resulting in a total cost of $1+1+2+2+2+2=10$. To further penalize entropy in each face we square the cost of each individual face, so the cost in the last example becomes $1^2+1^2+2^2+2^2+2^2+2^2=18$

- # Progress

  I've implemented Cube.py which contains a Rubik's Cube class which takes an array of face values to initialize from and has functions to

  1. Rotate any face
  2. Check if solved
  3. Compute cost for A*

4. generate a random cube

I've also defined the Search Problem and am using the UCS algorithm and priority queue provided in the Text Reconstruction homework problem.

I am testing my search algorithm by starting with a solved cube, randomly rotating some number of faces, and then seeing how long it takes for the algorithm to find the minimum number of turns to solve the cube. These are my results for regular UCS vs A* at different number or rotations from the solution.

1. UCS: Average 7 states explored (0.01s)
   A*: Consistent 2 states explored (0.002s)

2. UCS: Average 90 states explored (0.2s)
   A*: Consistent 4 states explored (0.002s)

3. UCS: Average 600 states explored (0.5s)
   A*: Consistent 10 states explored (0.01s)

4. UCS: Average 10000+ states explored (20s+)
   A*: Consistent 40 states explored (0.05s)

5. UCS: Average (MANY) states explored (too long)
   A*: Consistent 100 states explored (0.26s)

As you can see, A* significantly reduces the search space. At this point, I feel that I've mostly accomplished my original goal, and now feel that I should expand my original goal. I can keep making small improvements to the cost function to improve the A* performance, but I think it would be more interesting to extend my approach to have my algorithm learn solving algorithms instead of just searching for a solution. The motivation for this is that there are well known algorithms to solve Rubik's Cubes from any configuration, and given that I have now created a simple Rubik's Cube solver, a next step could be to use this tool to have a model learn these algorithms using it's own input and output as training data.