Part 2.1 Solving the Cube without Rotation Invariance

In this part, we implement the basic A* Search techniques to solve the 2 by 2 Rubik's Cube with spatial rotation invariance. There is only one goal state in this case: left with red, top with blue, right with green, front with orange, bottom with yellow, and back with purple. To solve the cube, each step has 12 different rotations: rotating six different faces with clockwise and counter-clockwise. The following section will explain the implementation of source code (see *mp_21.py*) to solve the Rubik's Cube.

Data Structure

The program reads input from text file and assigns each color to related position. Each face is represented by 1D list with index from 0 to 3 in the layout showing in the figure 1 below.

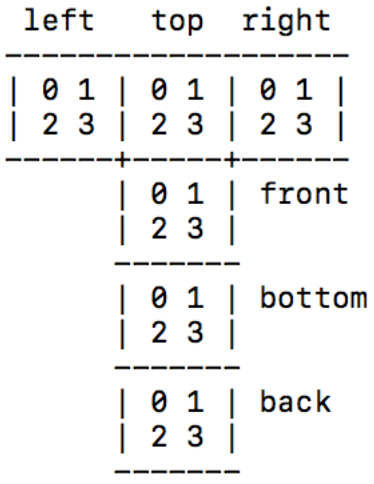


Figure 1

```
Rubiks Cube Data Structure

    cube = {
        'top':top,
        'bottom':bottom,
        'front':front,
        'back':back,
        'left':left,
        'right':right,
        'path':path
    }
```

Figure 2

As showing in figure 2, the Rubik's Cube is represented by a dictionary with seven elements: left, top, right, front, bottom, back, and path. Each face element stores the 1D color list and the path element stores the list of rotations applied.

Implementation

This algorithm use Priority Queue as frontier and stores the pair (distance, cube data structure) in each step. The distance is calculated by heuristic function which we will discuss after. The cube in priority queue will be sorted according to the distance. In each step, the algorithm will pop one cube with minimal distance from priority queue and check if the cube matches the goal state. If the cube reaches the goal state, the function will break the while loop and return the path, cost, and runtime. Otherwise, the algorithm will call the 12 different rotation functions and continue to the next loop. The pseudocode is showing below in figure 3. In each rotation function, the algorithm will push the new (distance, cube) pair to priority queue.

```
def solve:
    # use priority queue as frontier
    queue = PriorityQueue()
    # initialize the cube dictionary
    cube = {left, top, right, front, bottom, back, path}
    # find the distance with heuristic function
    distance = heuristic(cube)
    # initialize the cost
    cost = 0
    # push the start (distance, cube) pair
    queue.push((distance, cube))

    # recursion to solve the cube
    while queue is not empty:
        # get the newest cube
        cube_current = queue.pop()
        # break and print if reaches goal state
        if cube_current is cube_goal:
            print cube[path]
            print cost
            break
        # else apply 12 different rotations
        rotate_top_cw(cube_cur)
        rotate_top_ccw(cube_cur)
        rotate_bottom_cw(cube_cur)
        rotate_bottom_ccw(cube_cur)
        rotate_left_cw(cube_cur)
        rotate_left_ccw(cube_cur)
        rotate_right_cw(cube_cur)
        rotate_right_ccw(cube_cur)
        rotate_front_cw(cube_cur)
        rotate_front_ccw(cube_cur)
        rotate_back_cw(cube_cur)
        rotate_back_ccw(cube_cur)
```

Figure 3 pseudocode

Transition Model

The algorithm contains 12 different rotation translations. In each translation, the function takes the cube data structure as input and change the color according to the rotation. Each rotation will rotate the 4 colors in the face clockwise or counter-clockwise and rotate the 8 sides as well. The 12 different rotations has same algorithm and I will takes rotate_top_cw() as example. The figure 4 and figure 5 below will show the rotation.

```
   left    top    right
-----------------------
| l0 l1 | t0 t1 | r0 r1 |
| l2 l3 | t2 t3 | r2 r3 |
--------+-------+--------
        | f0 f1 | front
        | f2 f3 |
        ---------
        | b0 b1 | bottom
        | b2 b3 |
        ---------
        | k0 k1 | back
        | k2 k3 |
        ---------
```

Figure 4

rotation_top_cw()

```
   left    top    right
-----------------------
| l0 f0 | t2 t0 | k2 r1 |
| l2 f1 | t3 t1 | k3 r3 |
--------+-------+--------
        | r2 r0 | front
        | f2 f3 |
        ---------
        | b0 b1 | bottom
        | b2 b3 |
        ---------
        | k0 k1 | back
        | l3 l1 |
        ---------
```

Figure 5

After rotating the color, the algorithm will calculate the distance of new cube with heuristic(). This heuristic function calculates the distance between the current state and the goal state. However, the distance in A* search also needs the cost from start state to current state. In this case, the algorithm will add the distance from heuristic function to the length of path list which represents the number rotation applied. Then, the algorithm will push the new (distance, cube) pair to priority queue. In this part, we do not need to handle repeated state detection. Therefore, push the new cube directly to priority queue.

Heuristic Function

The heuristic function compares the current state and the goal state. Then calculate the number of misplaced cubes. I will discuss the algorithm and implementation first, and then discuss about the optimality and admissibility. As showing above, the algorithm stores the color in six different face lists and each list contains 4 colors in that face. In order to compare the states quickly, the algorithm will concatenate the 24 colors to string with order left, top, right, front, bottom, and back. Then comparing the color string with the goal state 'rrrrbbbbggggooooyyyypppp' and finding the number of misplaced cubes. However, the heuristic function only returns the distance from the current state to the goal state. In order to find the real distance in A* search, the algorithm needs to add the length of path list to the returned distance from this function. The pseudocode is showing below as figure 6.

```
# heuristic pseudocode
def heuristic(cube):
    # initialize the counter
    diff = 0
    # the goal state
    goal = 'rrrrbbbbggggooooyyyypppp'
    # comparing the cube and the goal char by char
    for i in range(24):
    if cube[i] is not goal[i]:
        # if char is not matched, diff++
        diff += 1
    return diff
```

Figure 6 pseudocode

However, in this case, the heuristic distance is neither optimal nor admissible, because the cost of each path is the constant value 1 but the number changes of the misplaced cubes can up to 8. After each rotation, the number of different colors on the rotated face will stay the same. However, the colors on the sides may have up to 8 changes. In this case, there are up to 8 changes of the misplaced cubes in each rotation. Therefore, if we divide the heuristic distance by the parameter larger than 8, the heuristic function will always return the value less than 1 and the length of the path will always have more priority. In this case, the solution with less steps will have smaller distance and push in the front of the priority queue.

The admissible solution is not necessary the optimal solutions. In this part, if the distance is divided by a parameter larger than 24, the algorithm will become common recursion with heuristic function that always return 0. The smaller parameter will give more weights to distance between the current state and the goal state, but the larger parameter will give more weights to the cost from start state to current state. The table below is the path, nodes expanded, and runtime of algorithm with parameter from 2 to 10.

Parameter		*Input Files*		
		Cube1_1.txt	Cube1_2.txt	Cube1_3.txt
1	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	303	314	338
	Runtime	0.245s	0.249s	0.291s
2	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	386	134	524
	Runtime	0.301s	0.107s	0.423s
3	**Path**	**T’ R’ Ba F’**	**F’ L L T’ F’**	**T’ F’ R R Bo’ Ba’**
	Nodes	**33**	**51**	**276**
	Runtime	**0.023s**	**0.040s**	**0.318s**
4	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	70	154	829
	Runtime	0.056s	0.132s	1.002s
5	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	52	97	718
	Runtime	0.044s	0.077s	0.587s
6	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	16	411	3163
	Runtime	0.011s	0.349s	2.759s
7	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	31	1129	10597
	Runtime	0.025s	0.971s	9.661s
8	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	90	1953	19091
	Runtime	0.073s	1.667s	19.445s
9	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	164	3559	34226
	Runtime	0.133s	3.044s	34.077s
10	Path	T’ R’ Ba F’	F’ L L T’ F’	T’ F’ R R Bo’ Ba’
	Nodes	303	7716	77331
	Runtime	0.245s	7.030s	82.713s

Figure 7

According to the table above, the parameter 3 is the value that makes the algorithm optimal. In this case, I believe that the value 3 is probably the average number of color changes in each rotations. The admissible solution should be solution with parameter more than 8, but the number of rotation is not too much in this part. The heuristic with parameter less than 8 can also get the solution with minimum steps. The solution is not unique. If we change the order of calling functions, the solution will be the different. I also got “Bo T T F’” for cube1_1.txt, “F’ R R Bo’ Ba’” for cube1_2.txt, and “T’ F’ R R Bo Ba’” for cube1_3.txt.

Question in Part 2.1

1. The number of misplaced cubes is not admissibly because there are at most 8 color changes in each rotation, but the cost of each rotation is just 1. In this case, the weights of the distance from the current state to the goal state is larger than the weights of cost from the start state to the current state. Therefore, we cannot make sure the length of path is minimum and we cannot make sure the heuristic is admissibly.
2. The heuristic should divide by a number larger than 8 to make it admissible. In this case, the weight of distance is always smaller than the weight of the path cost. The priority queue will select the state that has less path length.
3. The path, nodes expanded, and runtime are showing in the table above (figure 7).

Part 2.2 Adding Rotation Invariance

In this part, we add the rotation invariance and the rotations that are equivalent in spatial will be consider as the same. The most obviously difference in this part is that there are 24 goal states and the algorithm has repeated state detection. The rotations that have same spatial effects will be deleted. Source code in *mp_22.py*.

Modification of the Goal State

In this part, the goal state does not need to specify the color of the face. In this case, the algorithm only need to check if all 4 small cubes in each face have the same color. Therefore, if the elements in each face color list have the same character, the state reaches the goal state.

Additional Data Structure

In this part, the algorithm has one more data structure, the repeated state detection. This detection is represented by a hash table and each spatial unique state will be added to this hash table. The cube data structure in part 1 is represented by the dictionary. However, list and dictionary is unhashable and we need to convert the lists to a string before hashing.

Modification of Implementation

The repeated states detection is combined by two parts. In the first part, we will eliminate the number of functions called in each loop. In space, the rotation of the top and the bottom, the left and the right, and the front and the back are actually the same. For instance, rotating the top clockwise has the same effect as rotating the bottom clockwise. Therefore, we only need to call 6 different rotation function instead of 12 rotation functions.

```
rotate_top_cw(cube_cur)
rotate_top_ccw(cube_cur)
rotate_right_cw(cube_cur)
rotate_right_ccw(cube_cur)
rotate_front_cw(cube_cur)
rotate_front_ccw(cube_cur)
rotate_bottom_cw(cube_cur)
rotate_bottom_ccw(cube_cur)
rotate_left_cw(cube_cur)
rotate_left_ccw(cube_cur)
rotate_back_cw(cube_cur)
rotate_back_ccw(cube_cur)
```

Figure 8

```
rotate_top_cw(cube_cur)
rotate_top_ccw(cube_cur)
rotate_right_cw(cube_cur)
rotate_right_ccw(cube_cur)
rotate_front_cw(cube_cur)
rotate_front_ccw(cube_cur)
# rotate_bottom_cw(cube_cur)
# rotate_bottom_ccw(cube_cur)
# rotate_left_cw(cube_cur)
# rotate_left_ccw(cube_cur)
# rotate_back_cw(cube_cur)
# rotate_back_ccw(cube_cur)
```

Figure 9

reduce the function call

In the second part, we will check each state with previous states in hash table. In each rotation function, we will construct the 24 spatial equivalence state and check all 24 states with hash table. If any of those states appeared before, return the function without pushing the current state to priority queue. The code below is the pseudocode for repeated state detection in rotate_top_cw().

```
# construct the spatial equivalence states
cube_state_t = [None] * 24
# front faces front
cube_state_t[0] = cube_t['left'] + cube_t['top'] + cube_t['right'] + cube_t['front'] + cube_t['bottom'] + cube_t['back']
cube_state_t[1] = cube_t['bottom'] + cube_t['left'] + cube_t['top'] + cube_t['front'] + cube_t['right'] + cube_t['back']
cube_state_t[2] = cube_t['right'] + cube_t['bottom'] + cube_t['left'] + cube_t['front'] + cube_t['top'] + cube_t['back']
cube_state_t[3] = cube_t['top'] + cube_t['right'] + cube_t['bottom'] + cube_t['front'] + cube_t['left'] + cube_t['back']
# front faces top
cube_state_t[4] = cube_t['left'] + cube_t['front'] + cube_t['right'] + cube_t['bottom'] + cube_t['back'] + cube_t['top']
cube_state_t[5] = cube_t['bottom'] + cube_t['front'] + cube_t['top'] + cube_t['right'] + cube_t['back'] + cube_t['left']
cube_state_t[6] = cube_t['right'] + cube_t['front'] + cube_t['left'] + cube_t['top'] + cube_t['back'] + cube_t['bottom']
cube_state_t[7] = cube_t['top'] + cube_t['front'] + cube_t['bottom'] + cube_t['left'] + cube_t['back'] + cube_t['right']
# front faces back
cube_state_t[8] = cube_t['left'] + cube_t['bottom'] + cube_t['right'] + cube_t['back'] + cube_t['top'] + cube_t['front']
cube_state_t[9] = cube_t['top'] + cube_t['left'] + cube_t['bottom'] + cube_t['back'] + cube_t['right'] + cube_t['front']
cube_state_t[10] = cube_t['right'] + cube_t['top'] + cube_t['left'] + cube_t['back'] + cube_t['bottom'] + cube_t['front']
cube_state_t[11] = cube_t['bottom'] + cube_t['right'] + cube_t['top'] + cube_t['back'] + cube_t['left'] + cube_t['front']
# front faces bottom
cube_state_t[12] = cube_t['left'] + cube_t['back'] + cube_t['right'] + cube_t['top'] + cube_t['front'] + cube_t['bottom']
cube_state_t[13] = cube_t['top'] + cube_t['back'] + cube_t['bottom'] + cube_t['right'] + cube_t['front'] + cube_t['left']
cube_state_t[14] = cube_t['right'] + cube_t['back'] + cube_t['left'] + cube_t['bottom'] + cube_t['front'] + cube_t['top']
cube_state_t[15] = cube_t['bottom'] + cube_t['back'] + cube_t['top'] + cube_t['left'] + cube_t['front'] + cube_t['right']
# front faces left
cube_state_t[16] = cube_t['front'] + cube_t['top'] + cube_t['back'] + cube_t['right'] + cube_t['bottom'] + cube_t['left']
cube_state_t[17] = cube_t['front'] + cube_t['left'] + cube_t['back'] + cube_t['top'] + cube_t['right'] + cube_t['bottom']
cube_state_t[18] = cube_t['front'] + cube_t['bottom'] + cube_t['back'] + cube_t['left'] + cube_t['top'] + cube_t['right']
cube_state_t[19] = cube_t['front'] + cube_t['right'] + cube_t['back'] + cube_t['bottom'] + cube_t['left'] + cube_t['top']
# front faces right
cube_state_t[20] = cube_t['back'] + cube_t['top'] + cube_t['front'] + cube_t['left'] + cube_t['bottom'] + cube_t['right']
cube_state_t[21] = cube_t['back'] + cube_t['left'] + cube_t['front'] + cube_t['bottom'] + cube_t['right'] + cube_t['top']
cube_state_t[22] = cube_t['back'] + cube_t['bottom'] + cube_t['front'] + cube_t['right'] + cube_t['top'] + cube_t['left']
cube_state_t[23] = cube_t['back'] + cube_t['right'] + cube_t['front'] + cube_t['top'] + cube_t['left'] + cube_t['bottom']

# check all 24 states
cube_state_str_t = [None] * 24
    for i in range(24):
        cube_state_str_t[i] = ''.join(cube_state_t[i])
        # return if the state is repeated
        if cube_state_str_t[i] in state:
            return
```

Figure 10 pseudocode

The repeated state detection will reduce the redundant function call and also reduce the states in priority queue and avoid redundant nodes expanded and runtime in each function call. In the code above, we concatenate the states as string and add the string to hash table.

Modification of Heuristic Function

With spatial rotation, the distance between the current state and the goal state is not restricted to the distance to the only goal state as part 1. In this part, the spatial heuristic function should

consider 24 different goal states and return the minimum distance to one of the goal states. Therefore, we modified the heuristic function by adding 23 more goal states and return the minimum distance. The figure 11 below shows the pseudocode for heuristic function.

This heuristic function also need to divide a parameter to make it admissible. The modification only changes the number of the goal states and the algorithm will select the shortest distance between the current state and one of the goal states. In this case, the maximum number of color changes for each rotation is still 8. Therefore, we still need to divide the distance by a number more than 8 to make the heuristic function admissible.

```
def heuristic(cube_h):
    # list to store disatance to 24 different goal states
    diff = [0] * 24
    # list to store 24 goal states
    goal = [None] * 24
    # with front faces front
    goal[0]= ['r','r','r','r','b','b','b','b','g','g','g','g','o','o','o','o','y','y','y','y','p','p','p','p']
    goal[1]= ['y','y','y','y','r','r','r','r','b','b','b','b','o','o','o','o','g','g','g','g','p','p','p','p']
    goal[2]= ['g','g','g','g','y','y','y','y','r','r','r','r','o','o','o','o','b','b','b','b','p','p','p','p']
    goal[3]= ['b','b','b','b','g','g','g','g','y','y','y','y','o','o','o','o','r','r','r','r','p','p','p','p']
    # with front faces top
    goal[4]= ['r','r','r','r','o','o','o','o','g','g','g','g','y','y','y','y','p','p','p','p','b','b','b','b']
    goal[5]= ['g','g','g','g','o','o','o','o','r','r','r','r','b','b','b','b','p','p','p','p','y','y','y','y']
    goal[6]= ['b','b','b','b','o','o','o','o','y','y','y','y','r','r','r','r','p','p','p','p','g','g','g','g']
    goal[7]= ['y','y','y','y','o','o','o','o','b','b','b','b','g','g','g','g','p','p','p','p','r','r','r','r']
    # with front faces back
    goal[8]= ['r','r','r','r','y','y','y','y','g','g','g','g','p','p','p','p','b','b','b','b','o','o','o','o']
    goal[9]= ['b','b','b','b','r','r','r','r','y','y','y','y','p','p','p','p','g','g','g','g','o','o','o','o']
    goal[10] = ['g','g','g','g','b','b','b','b','r','r','r','r','p','p','p','p','y','y','y','y','o','o','o','o']
    goal[11] = ['y','y','y','y','g','g','g','g','b','b','b','b','p','p','p','p','r','r','r','r','o','o','o','o']
    # with front faces bottom
    goal[12] = ['r','r','r','r','p','p','p','p','g','g','g','g','b','b','b','b','o','o','o','o','y','y','y','y']
    goal[13] = ['g','g','g','g','p','p','p','p','r','r','r','r','y','y','y','y','o','o','o','o','b','b','b','b']
    goal[14] = ['y','y','y','y','p','p','p','p','b','b','b','b','r','r','r','r','o','o','o','o','g','g','g','g']
    goal[15] = ['b','b','b','b','p','p','p','p','y','y','y','y','g','g','g','g','o','o','o','o','r','r','r','r']
    # with front faces left
    goal[16] = ['o','o','o','o','r','r','r','r','p','p','p','p','b','b','b','b','g','g','g','g','y','y','y','y']
    goal[17] = ['o','o','o','o','y','y','y','y','p','p','p','p','r','r','r','r','b','b','b','b','g','g','g','g']
    goal[18] = ['o','o','o','o','g','g','g','g','p','p','p','p','y','y','y','y','r','r','r','r','b','b','b','b']
    goal[19] = ['o','o','o','o','b','b','b','b','p','p','p','p','g','g','g','g','y','y','y','y','r','r','r','r']
    # with front faces right
    goal[20] = ['p','p','p','p','r','r','r','r','o','o','o','o','y','y','y','y','g','g','g','g','b','b','b','b']
    goal[21] = ['p','p','p','p','b','b','b','b','o','o','o','o','r','r','r','r','y','y','y','y','g','g','g','g']
    goal[22] = ['p','p','p','p','g','g','g','g','o','o','o','o','b','b','b','b','r','r','r','r','y','y','y','y']
    goal[23] = ['p','p','p','p','y','y','y','y','o','o','o','o','g','g','g','g','b','b','b','b','r','r','r','r']

    # check the distance for each goal states
    for i in range(24):
        for j in range(24):
            if cube_h[j] is not goal[i][j]:
                diff[i] += 1
    # return the minimum distance and divide by parameter
    return min(diff)/parameter
```

Figure 11 pseudocode

Performance

With the repeated state detection and the modified heuristic function, the performance of the algorithm improved by a great amount. The table below shows the path, nodes expanded, and runtime for the input files from part 2.2 as well as the input file from part 2.3.

		Input Files			
Parameter		Cube2_1.txt	Cube2_2.txt	Cube2_3.txt	Cube3_1.txt
1	Path	T' R'	F' T' R' R' T' R'	F' R' T T R'	F R' T R F' R F
	Nodes	3	2343	126	956
	Runtime	0.0035s	3.080s	0.165s	1.260s

2	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	3	207	128	1309
	Runtime	0.0027s	0.265s	0.169s	1.682s
3	**Path**	**T’ R’**	**F’ T’ R’ R’ T’ R’**	**F’ R’ T T R’**	**F R’ T R F’ R F**
	Nodes	**3**	**82**	**61**	**576**
	Runtime	**0.0032s**	**0.105s**	**0.083s**	**0.744s**
4	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	4	143	100	1340
	Runtime	0.0039s	0.185s	0.136s	1.723s
5	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	3	251	72	902
	Runtime	0.0029s	0.322s	0.101s	1.165s
6	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	4	639	142	2091
	Runtime	0.0039s	0.851s	0.188s	2.682s
7	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	4	840	198	2760
	Runtime	0.0060s	1.090s	0.262s	3.676s
8	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	5	1539	379	4958
	Runtime	0.0073s	1.991s	0.487s	6.399s
9	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	4	2518	431	7935
	Runtime	0.0051s	3.270s	0.559s	11.057s
10	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	4	2598	439	8211
	Runtime	0.0045s	3.333s	0.573s	10.908s
15	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	10	4574	822	13645
	Runtime	0.0154s	5.935s	1.051s	21.021s
20	Path	T’ R’	F’ T’ R’ R’ T’ R’	F’ R’ T T R’	F R’ T R F’ R F
	Nodes	13	10381	1093	13645
	Runtime	0.0150s	13.508s	2.431s	21.021s

Figure 12

According to the table above, the algorithm reaches optimal when parameter is 3 and reaches admissible only when parameter larger than 8. However, the number of steps are not too large and the heuristic function with parameter under 8 are all admissible. The solution is combined by only F, T, and R, because we reduce the function call for Ba, Bo, and L. The number of nodes expanded is the number of states popped from the priority queue. I also wrote a WebGL animation for all input files. Please check the folder *Animation* and open the *index.html*.

Comparing Cube1_1.txt and Cube1_2.txt
The solution for Cube1_1.txt is ***T' R' Ba F'*** while the solution for Cube2_1.txt is ***T' R'***. We can conclude that the first two steps already solved the Rubik's Cube and the next two rotations ***Ba F'*** changes the direction of the faces.

The performances of the two algorithm have huge difference. The algorithm in part1 explored 33 nodes when parameter is 3 and the minimum nodes expanded is 16 when parameter is 6. While in the part2, the algorithm only explored 3 nodes when parameter is 6. The difference caused by several reasons.

First, the number of function call are different. In the first algorithm, there are 12 different rotation functions and the solution requires 4 steps. In this case, the number of nodes pushed to the priority queue is at least $12^4 = 20736$. Although we are using priority queue, there are still numerous states that have same priority. In the second Algorithm, there are only 6 different rotation functions and the solution only contains 2 steps. In this case, the number of nodes pushed to the priority queue is $6^2 = 36$ states. With priority queue, the node explored will even be less.

Second, the repeated states detection will remove the redundant states. The number of the states pushed to the priority queue in algorithm 2 is much less than the number of algorithm 1. The repeated states detection will further enlarge the difference. After removing redundant states, there are much less states that have the same priority. The nodes expanded in algorithm 2 will be less.

Third, the goal states are different. The algorithm 1 has only one goal states while the algorithm 2 has 24 goal states. The states in algorithm 2 are much easier to reaches the goal states. Therefore, the algorithm 2 has much better performance than the algorithm 1.

Part 2.3 Extra Credit

The result of input Cube3_1.txt is showing above in figure 12. The solution is ***F R' T R F' R F***, the number of nodes expanded is 576, and the runtime is 0.744s. The solution will be guaranteed admissible when parameter is larger than 8. According to the table, we can find that the heuristic function with parameter 3 got the same solution as admissible heuristic function. Therefore, the solution is admissible.

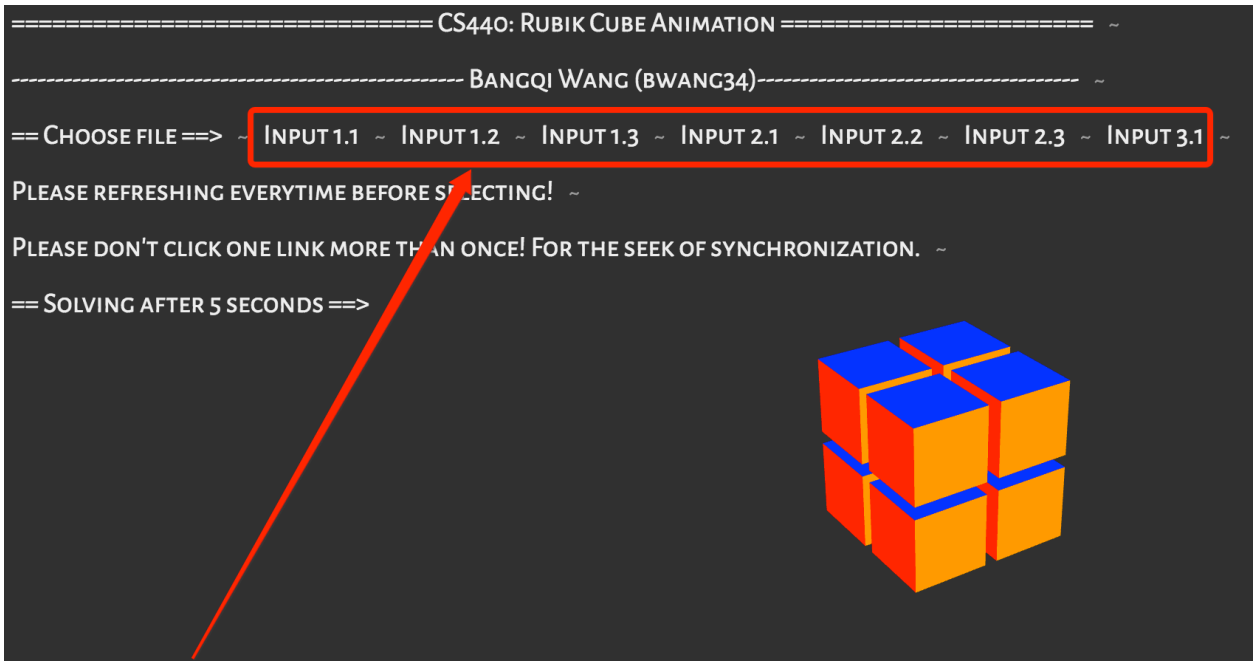
According to the wiki page *https://en.wikipedia.org/wiki/Pocket_Cube*, The maximum number of steps to solve a 2x2 Rubik's Cube is 14 steps.

> Any permutation of the eight corners is possible (8! positions), and seven of them can be independently rotated (3^7 positions). There is nothing identifying the orientation of the cube in space, reducing the positions by a factor of 24. This is because all 24 possible positions and orientations of the first corner are equivalent due to the lack of fixed centers. This factor does not appear when calculating the permutations of N×N×N cubes where N is odd, since those puzzles have fixed centers which identify the cube's spatial orientation. The number of possible positions of the cube is
>
> $$\frac{8! \times 3^7}{24} = 7! \times 3^6 = 3,674,160.$$
>
> The maximum number of turns required to solve the cube is up to 11 half or quarter turns, or up to 14 quarter turns only.[2]

From *https://en.wikipedia.org/wiki/Pocket_Cube*

3D Animation

I also wrote a 3D WebGL animation for all input files. The animation is in the folder *Animation* and open *index.html* in browser. The image below is the epitome of my animation.



Click the input files and wait 5 seconds for solving. You can drag the image to view other faces. Please do not click the link more than once. It is a hardcoding javascript animation and I don't have enough time to handle the synchronization issues. Please do not click one link more than once and please refresh the page before choosing new input. If you meet any bug, please refresh the website.

The basic layout comes from github https://github.com/jwhitfieldseed/rubik-js. However, I changed almost 80% of the codes to fit our needs.

Good Luck! Have Fun!

Attribution

I am Bangqi Wang (bwang34). My teammate is Yifei Li (yifeili3).

I wrote starter version of BFS and Greedy search for part 1.1 and the entire part 2, including the extra. I also hardcoding the WebGL animation showing above. This report concentrates on part 2 Rubik's Cube.

Yifei tested my BFS and Greedy Code and also handle the rest of part 1. His report will focus on the first part, Maze Solver.