

CAB203 Project Report

Johnny Madigan
N10027190
June 2021

1 Introduction

Since the beginning of mankind, we have always been intrigued by puzzles because of the challenge they pose and our desire to solve things. Fast-forward to now, and certain puzzles have become infamous through their complexity and deceiving looks. One such puzzle is the Pocket Cube, smaller cousin to the iconic Rubik's Cube.

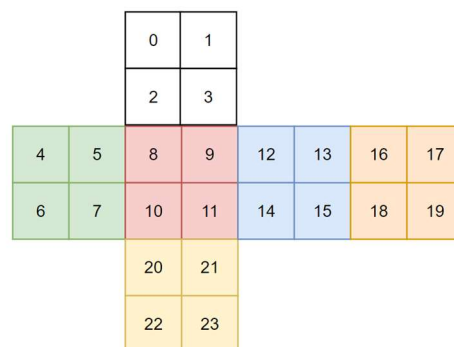
The challenge comes from twisting and turning the cube until each side is one colour. From its fame, the internet has been flooded with algorithms so that anyone can memorise and solve the Pocket Cube themselves. However, these algorithms are designed as easy patterns that anyone can remember to solve the cube in the shortest amount of time, not the least moves.

This is where our puzzle comes in. No matter what scrambled Pocket Cube we are given, we must figure out the smallest number of moves required to solve it. We will define moves as quarter turns only.

2 Instance model

Since we are approaching this from a logically, mathematical, and computer-based point of view, we need to encode any scrambled cube as an instance that we can manipulate. Our instance needs to be a *Python* object where we can easily pass all the information as one argument.

The object instance will be a *string* of 24 characters, which will represent the 24 stickers (4 stickers on each face). See the diagram below for a visual of how we will begin translating a Pocket Cube into a *Python* object.

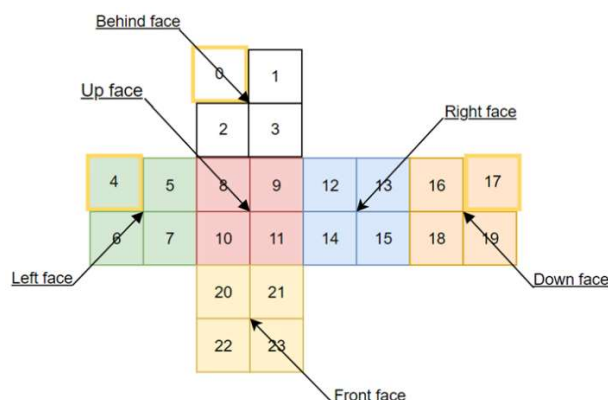


W white **R** red **Y** yellow **G** green **B** blue **O** orange

After laying out any real-life pocket cube like the diagram above, you can follow the order of the numbers and note their sticker's colour to translate your cube into the instance model. For example, the solved cube will look like this:

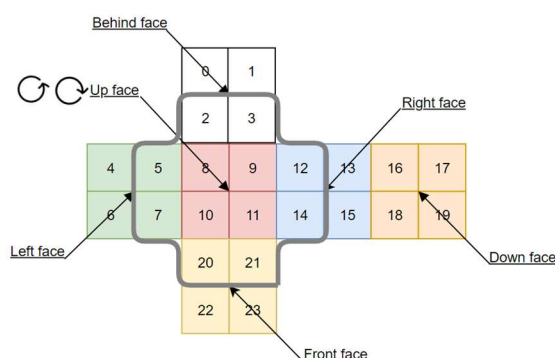
```
testData = "WWWWG GGGRRRRBBBBOOOOYYYY"
```

Research shows us that a Pocket Cube has 3,674,160 possible permutations (*"Pocket Cube"*, n.d.). We can validate this instance if it can be changed into over 3.6 million permutations. Briefly breaking down how this number was found; we start with a fixed corner as the pivot, which will be the corner for the BEHIND, DOWN and LEFT faces (letters at positions 2,5,17).

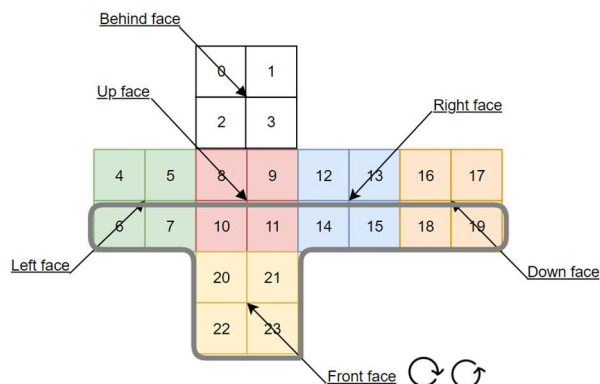


Highlighted are the stickers that make up one of the 8 corners when folding the cube back

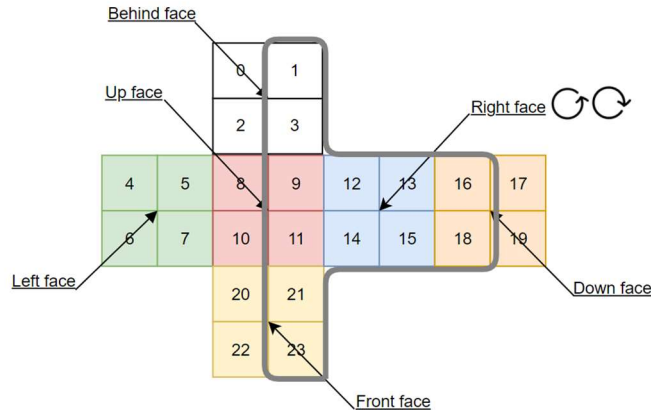
From here there are only 6 new permutations we can make from quarter turns. Each move rotates a face (4 corners clockwise or anti-clockwise) and only 7 of the corners can be rotated as one corner is the fixed pivot corner. The diagrams below show which stickers get rotated when doing the 6 quarter turns, which will rearrange the instance.



Rotating the 'UP FACE' clockwise & anti-clockwise



Rotating the 'FRONT FACE' clockwise & anti-clockwise



Rotating the RIGTH FACE adjacent to the front, clockwise & anti-clockwise

This means the problem instance is capable of 7 factorial permutations, which is 5040 permutations.

$$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040$$

Next, each corner can be oriented in 3 ways by rotating each sticker on it which will be 3^6 more orientations. The math behind this is that with the pivot corner's orientation never changing, we only need to find the orientation for 6 other corners to automatically know the final corner's orientation. As each corner moves (group of 3 colour stickers), so will the stickers positions. This is only possible by having 24 characters in the instance, rather than 8 for each corner only, or 6 for each face.

Finally, we multiple these corner orientations with all the permutations to get the final number of possible permutations.

$$7! * 3^6 = 3,674,160$$

By breaking this down you can now see how the instance is capable of over 3.6 million permutations just like a real-life Pocket Cube.

3 Solution model

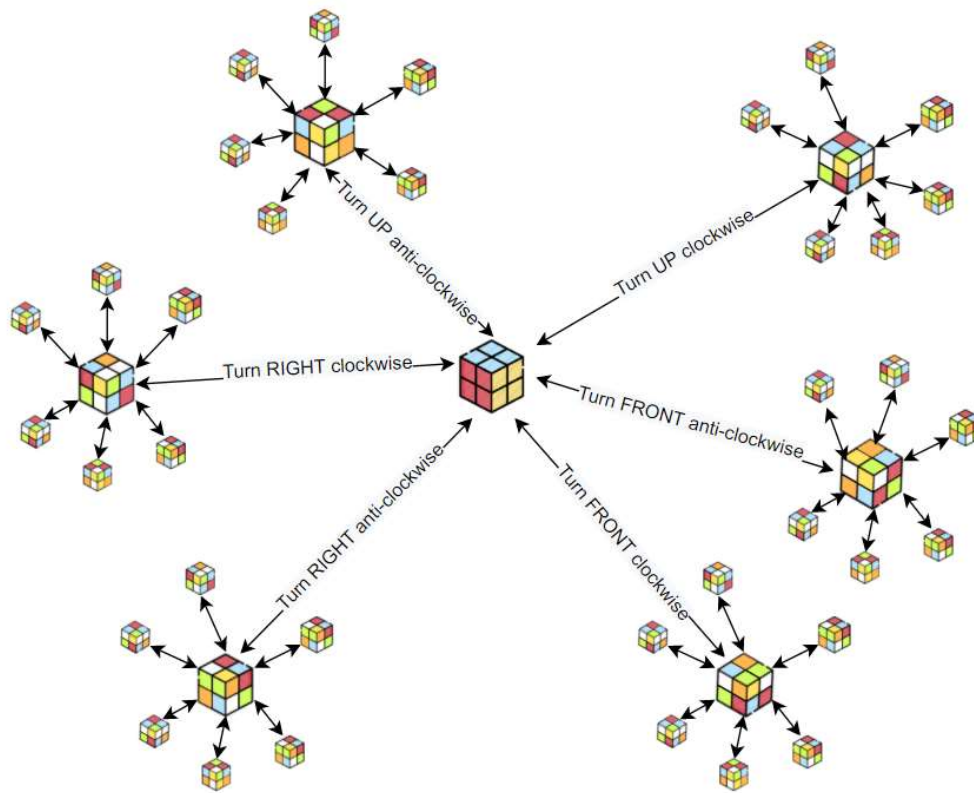
The solution is the number of minimal steps required to solve the cube. This will always be a whole number, as we might get 0 steps if the problem cube is already solved.

This solution model will come from the index of the distance class that contains the problem instance in a *Python list*. This list of distance classes will start with the solved instance, and the next class will hold all new instances after 1 legal move, the next after 2 legal moves and so on. This is how we can easily see how many steps it took to get to the problem instance from the solved cube. In other words, the minimum number of steps it would take to solve the problem instance.

4 Problem model

Given a scrambled cube encoded as the specified instance, we will treat this as the vertex to find. We will start by generating a *Cayley Graph* where each vertex will be modified through 6 unique quarter turns. There are 6 moves rather than 12, as half the moves are redundant since we do not need to consider the orientation as stated in section 2.

After applying any of these 6 moves to a vertex, the graph will generate a new vertex along with an edge between the two. The diagram below shows a small version of the *Cayley Graph*, which will quickly grow up to a diameter of 14. This diameter is known is due to the fact that all Pocket Cubes are solvable within 14 quarter turns or less, assuming the cube was scrambled legally (*"God's Number"*, n.d.).



The partial Cayley Graph for the Problem Model

In mathematical notation, this graph will look like:

$$G = (V, E)$$

And the generator will be the set S , which will contain each of the 6 quarter turns:

$$S = \{U, U', F, F', R, R'\}$$

Where V is the set of cube vertices generated along the *Cayley graph* branching out from the solved cube instance. Meanwhile, E is the set containing the edges between two cubes that can become each other through a single quarter turn.

v_0 is the solved cube a.k.a. the root vertex:

$$v_0 = root$$

The mathematical notation for the generating a new vertex is:

$$v_m = v_n * s : s \in S, v_n \in V$$

After a new vertex (v_m) is generated, it is added to V if the new vertex is not already a member of V :

$$v_m \notin V$$

After a new vertex (v_m) is generated, there will be 2 edges going each way as the graph is undirected, and added to E if the relationships do not already exist in E :

$$(v_m, v_n) \text{ and } (v_n, v_m) \notin E$$

5 Solution method

To begin, we must verify that the problem instance is solvable. We start by making the instance *string* object uppercase and sorting characters alphabetically, also for the solution instance. If the strings are identical, the contents are verified. However, unlike the solved instance, which we defined to have a pivot corner fixing its orientation (3 characters at 3 specific indexes in the *string* object). The problem instance can be oriented in 24 different ways in 3D space depending on the user's input. There are 6 orientations where each face takes a turn being face-down. You can then twist the whole cube 90 degrees clockwise 4 times while keeping the same face down. The simplest way to tackle this variable is to rotate the problem instance in these 24 unique orientations. If this solution method finds any of these orientations for the problem instance, the problem is solved.

The *Cayley Graph* defined in section 4 will always start with the solved cube as the root vertex.

$$v_0 = root$$

It will then use an algorithm to continuously rotate the cubes in all the ways specified in the *Instance Model* from section 2. This moves set consisting of 6 unique quarter turns also corresponds to the generating set S from section 4.

$$S = \{U, U', F, F', R, R'\}$$

Furthermore, the mathematical notation for the generator from section 4 defines how and when a new vertex is added to the *Cayley Graph*, along with its edges:

Generator:

$$v_m = v_n * s : s \in S, v_n \in V$$

New vertex from generator is added to the graph when:

$$v_m \notin V$$

New corresponding edges from the generator are added to the graph when:

$$(v_m, v_n) \text{ and } (v_n, v_m) \notin E$$

This generator will run in a loop until any orientation of the problem instance is added to the *Cayley Graph*. Once found, we will then use *Breadth-first search* to find the shortest path between the solved cube and the problem cube in the graph. Before we do this however, we will simplify the entire graph down into only a few groups, which will be distance classes branching out from a root vertex.

Starting with the solved cube as the root vertex, the adjacent distance class will contain all cubes (vertices) that can be created from the solved cube (root vertex) with only one quarter turn. The next class will contain all cubes that can be created from the solved cube in only two quarter turns, ignoring all previously assigned cubes, and the pattern continues.

We will calculate the distance classes recursively. We start by assigning all our vertices to the set V_0 representing the initial set of vertices that are yet to be assigned a distance class.

$$V_0 = V$$

Since the solved cube is the root vertex v_0 (lowercase v for vertex, not to be confused with uppercase V for the set of vertices), it will be the sole member of the first distance class. Just like the initial set of vertices V_0 we will also have an initial distance class D_0 . We will set the initial distance class to only contain the root node.

$$D_0 = v_0$$

Each time the function calls upon itself in a recursive loop, there will be new iterations of V_0 denoted as V_{new} or V_j where the vertices that have already been assigned a distance class are culled from the set. Leaving us with only the vertices that have not been touched yet in V_j

To remove all the old vertices we have already seen, we will find the set difference between the vertices in the previous V_j which will be V_{j-1} and the vertices in the previous distance class D_{j-1}

Therefore, the next V_j containing ONLY the vertices that have not been assigned a class yet will be:

$$V_j = V_{j-1} \cap D_{j-1}$$

After removing all old vertices above, the next step is to find the next distance class. Which will be achieved by sorting through the new set of vertices that have not been assigned a class yet V_j and finding all vertices connected by an edge to the previous distance class D_{j-1}

The mathematical notation for finding this neighbourhood (adjacent vertices) for the next distance class is:

$$N_g(S) = \{v \in V : (u, v) \in E, u \in S\}$$

Where V will be the new set of vertices yet to be assigned a class $= V_j$

Where E will be the set of edges in our graph from earlier $= E$

Where S will be the old distance class vertices $= D_j$

Then we repeat until completed, finally returning this set of distance classes to the Print Solution function. The Print Solution function will do the *Breadth-first search* starting from the first class (which contains only the solved cube). The path will be searching through all vertices in one class before moving onto the next class in order. This means all cubes that take a minimum of 1 step to solve are searched first, then all the cubes that take a minimum of 2 steps to solve and so on.

This process of elimination is what makes *Breadth-first search* perfect for finding the shortest path. Once the search finds the problem instance, it will print the index of the distance class that the problem instance is in. This is because the index corresponds to the number of quarter-turns it took to get from the solved cube to the problem cube, or vice versa... a.k.a. the minimum number of steps it takes to solve the cube.

In summary the process of the solution method is:

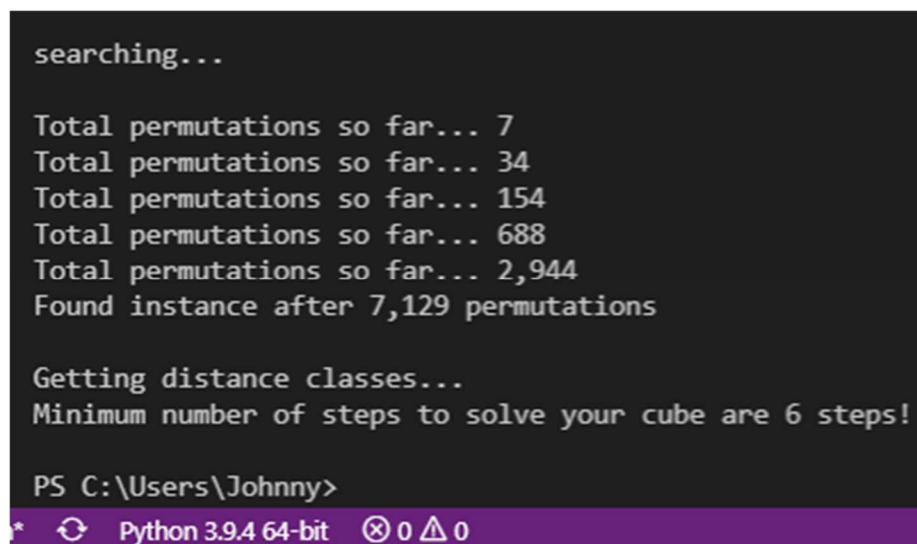
- Get all 24 possible orientations for the instance
- Generate the *Cayley Graph* starting from the solved cube
- Once one of the orientations for the instance is found, generate the distance classes recursively
- Perform a *Breadth-first search* to find the shortest path between the solved cube and the instance and print the index (a.k.a. the minimum number of steps it takes to solve)

```
searching...

Total permutations so far... 7
Total permutations so far... 34
Total permutations so far... 154
Total permutations so far... 688
Total permutations so far... 2,944
Found instance after 7,129 permutations

Getting distance classes...
Minimum number of steps to solve your cube are 6 steps!

PS C:\Users\Johnny>
```

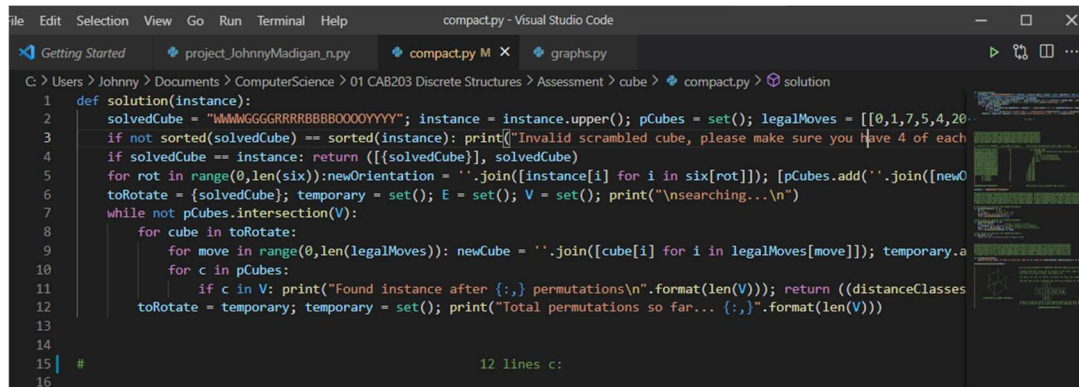


References

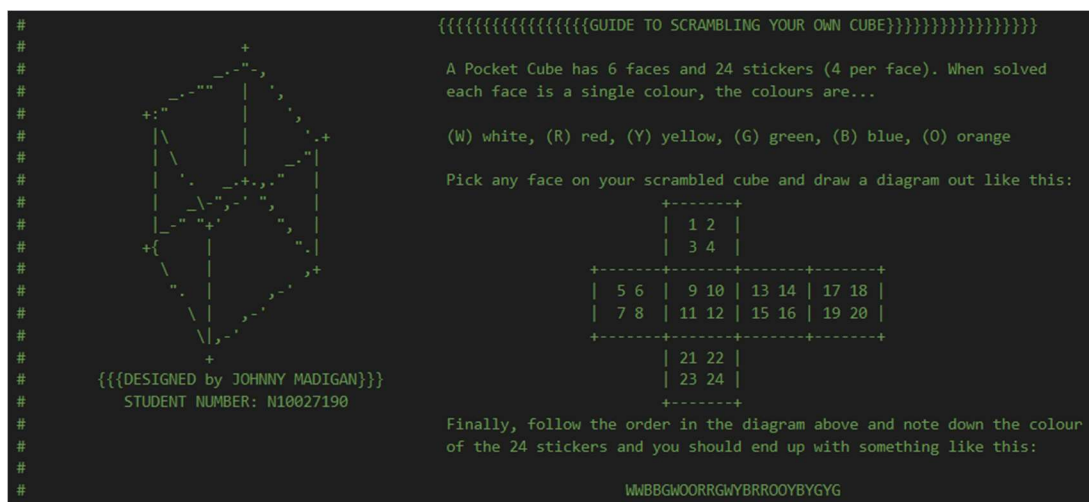
Pocket Cube. (n.d.). Retrieved from [God's Number - Looking for the optimal Rubik's Cube solution \(ruwix.com\)](http://ruwix.com/Gods-Number-Looking-for-the-optimal-Rubik-s-Cube-solution/)

God's Number. (n.d.). Retrieved from [Pocket Cube - Wikipedia](https://en.wikipedia.org/wiki/Pocket_Cube)

As stated by Matt our Unit Coordinator, all project topics were designed to be solved in under 20 lines of code. Although completely unreadable, by removing all comments, using list comprehension, short-hands and one-liners... the solution can be one reduced to 12 lines!

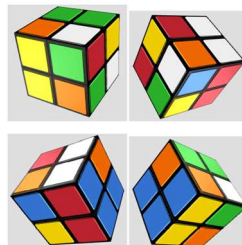


This was an extra block of code I put in for a more user-friendly experience:



The test data from the Section 6 of the assignment brief PDF was constructed by making a diagram of the cube (see above screenshot) after analyzing the pictures given. To verify, I used the [website given \(grubiks.com\)](https://grubiks.com), and painted the same cube to solve it. Both my solution method and the website solved the cube in 6 quarter moves.

```
testData = "OB BBWRORRBYYYOGOGWGWGYR"
```



From Section 6: Test data