Charles Nguyen

COMP 1405 Z – Intro to Computer Science I

Project Analysis Report

A graph theory implementation of the relationship between group theory in abstract mathematics and harmonious chords in music

I. Project Problem

Develop an algorithm and data structure to store and solve a 2x2x2 Rubik's cube optimally in terms of memory and time complexity. God's Number is used to specify the greatest number of moves required to solve a Rubik's cube. For a 2x2x2, it has been proven to be 11. The algorithm will be required to solve the cube in at most 11 moves and execute very fast (ideally less than 2s)

Subsequently, design an algorithm and data structure to assign music notes to the cube's faces and play the notes as the cube get twisted and solved. The final goal is to produce sequences of chords or notes that sounds pleasant.

II. Background Information

We first introduce some concepts regarding what a Rubik's cube and a chord is. Knowing this will help us understand better what the problem is talking about. Most importantly the concepts help us forming a concrete approach to solving the problem, as well as designing and implementing our algorithms/data structure.

1. 2x2x2 Rubik's cube (Group Theory)

In group theory, a group is basically a set containing elements with an arbitrary binary operator that can be performed on these elements. These elements and operator can go along with each other as long as certain conditions (axioms) must be met. We will only focus on 3 of them that can help solving the cube, which are:

- Associativity: Say we have 2 numbers a, b and an addition operator, then $a + b = b + a$
- Identity element: This is equivalent to the number 0 since $a + 0 = a$ for all a
- Inverse element: Think of this as negative number of $a, -a$, such that $a + (-a) = 0$ which is the identity element. An inverse undo what the original has done.

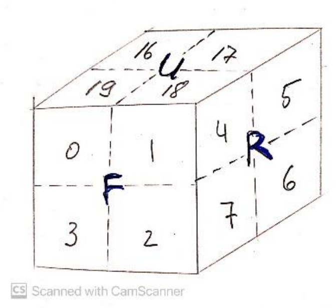
When you manipulate a Rubik's cube, you will form a Rubik's cube **group** whose elements are every possible configurations/states of the cube.

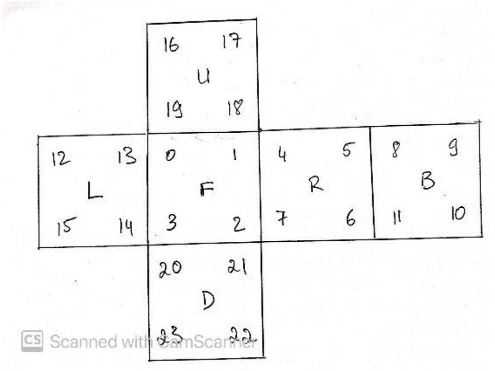
Charles Nguyen

A 2x2x2 Rubik's cube, more commonly referred as a Pocket cube, is made up of smaller cubes called cubies (there are $2x2x2 = 8$ of them), and each has 3 faces outward (thus there are $8x3 = 24$ small cubie's faces in total).

The Pocket cube as a whole has 6 larger faces which are Front, Left, Right, Up and Down. To play this cube, i.e. to scramble or solve it, we move/twist these faces. You can rotate it quarterly 90^0 clockwise, 90^0 counterclockwise or half 180^0 clockwise. Applying these 3 basic moves to 6 faces and we have a total of 18 moves to choose. But due to the structure symmetry, there are moves equivalent to each other (i.e. rotating the Front face clockwise = rotating the Back face counterclockwise, etc. …). This is following the **Associativity** axiom of a group. We can therefore reduce to a total of only 9 moves now with only 3 faces to consider. The project involves with Front, Right and Up faces.

In standard notation, when we apply a move we create a new configuration called permutation of the Pocket cube, and applying a move is permuting the cube. A colorful Rubik's cube is more familiar with everyone, but to have an algorithm that works with the cube, we represents the cube with something more discrete which is integers. We label each small cubie's face with a number from 0-23 (since us computer scientists like 0-base indexing). We are indexing the cubie's faces and in doing so we have the following:





When we move a face, 4 cubies move. In other word, the respective cubies' faces (12 of them) get moved. It makes sense that if a cubie is moved, the 3 small cubie's faces attached to it have to move along as well and always stay together attached to that cubie. Take an example when the Front face is rotated 90^0 clockwise. On the Front face, 3 goes (maps) to 0; 0 goes (maps) to 1; 1 to 2 and 2 back to 3. But other cubie's faces that attached to the 4 cubies must move along as well. We have 20 to 13, 13 to 18, 18 to 7 and 7 back to 20. Likewise, 14 goes to 19, 19 to 4, 4 to 21 and 21 back to 14. We notice some form of a cycle here, and this is called cyclic permutation which is a part of group theory also and denoted as follow:

$$F = \begin{pmatrix} 0 & 3 & 2 & 1 & 19 & 14 & 21 & 4 & 13 & 20 & 7 & 18 \\ 3 & 2 & 1 & 0 & 14 & 21 & 4 & 19 & 20 & 7 & 18 & 13 \end{pmatrix}$$

These are only the cubie's faces that got moved by the Front face twist; others that are unaffected can be thought of as mapping to itself. We rewrite such that we have a full 24 cubie's faces and arrange such that the top row is sorted in ascending order:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 3 & 0 & 1 & 2 & 19 & 5 & 6 & 18 & 8 & 9 & 10 & 11 & 12 & 20 & 21 & 15 & 16 & 17 & 13 & 14 & 20 & 21 & 22 & 23 \end{pmatrix}$$

Here, the top row is the original configuration of the cube (more precisely the index of the original configuration), and the bottom row is the move you apply results in. To conclude, this can be rephrased in words as: Within the cube, the 0^{th} face becomes the 3^{rd} face; the 1^{st} face becomes the 0^{th} face; the 2^{nd} face becomes the 1^{st} face; etc. … when applying the Front move. This is only a 90^0 clockwise move, there are counterclockwise and 180^0 moves, as well as applying these for the 2 remaining Pocket cube's faces as well.

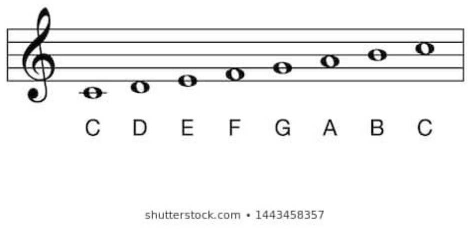
To reduce our workload of writing out/hard coding these move configurations (which can easily create bugs), we can avoid writing for the counterclockwise move by using the **Inverse** group axiom. Notice that counterclockwise move undo what the clockwise move did before, that is moving Front face clockwise, then do a Front face counterclockwise will just give us back the original configuration (**Identity**) of the cube. To find an inverse of a clockwise move, from the cyclic permutation notation above, we flip the 2 rows then arrange such that the top row is sorted in ascending order. As exactly like before, the top row is the original configuration of the cube and the bottom row is the move you apply, which is now the inverse of a clockwise move.

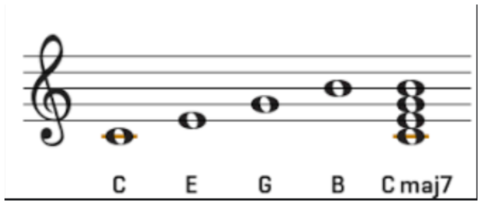
2. Harmonious Chords (Music Theory)

When certain notes are grouped together, they form a chord which sounds harmonious and pleasant to our ears. A chord typically consists of 3 to 4 notes, and in this project a 4-note chord, also called a 7th chord, will be used since each face of the Pocket cube has 4 small cubie's faces.

On a given music (key) scale, start from the root note of that scale and from there stack up the notes so that we have a chord that looks like a snowman. For example, in a key of C Major scale:

Charles Nguyen



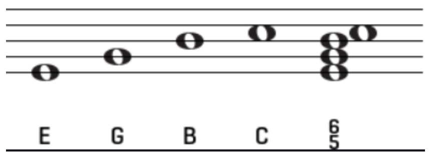


The root note of C major scale is the note C. Build a 7th chord by stacking up the 1st, 3rd, 5th and 7th notes on top of each other (like a snowman) and you have (C, E, G, B) which is a C major 7th chord.

Multiple chords can be used together to create a chord progression that eventually makes up a song/composition, creates tension/rest, etc. … Our project only involves with 3 Pocket cube's faces to solve so we need to find the right 7th chords that can form a 3-chord progression. Luckily, we found one and it is called the 2-5-1 chord progression, typically appears in Jazz.

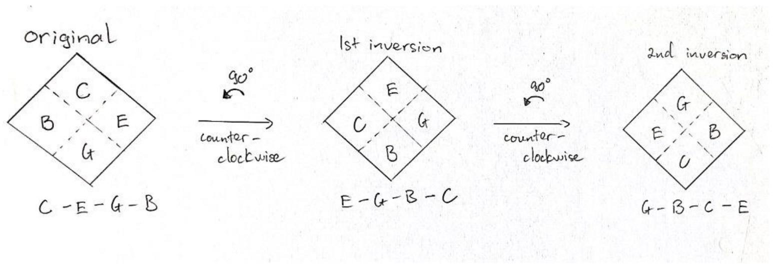
The numbers 2, 5 and 1 refers to the root note of the chords in music scale. In a C major scale above, the 2nd note is D which is the root note to create a 7th chord upon it. Continue to stack up the chords from note D, and we have (D, F, A, C) which is a D minor 7th chord. Likewise with the 5th note G, and from there we have chord (G, B, D, F) which is a G dominant 7th. The 1st note is C which is what we found above. Put these together and we play the chords in the order 2-5-1: the D chord, G chord and finally the C chord. There are no less than 20 keys/music scales in total and each time we apply this chord progression to a different scale we have a completely new chords formed.

An idea we also make use of in music theory is chord inversion, which ties a bit to group theory as well. Our music scale and even our chords can be considered as a group whose elements are only those notes in that scale or chord. In a music scale, if we want to continue with our scale going up, we would still come back to the root note and stay in that scale with those same notes, only this time the notes are an octave higher, meaning they have higher pitch (sound 'higher'). Likewise, when you invert a chord, meaning putting the bottom root note on top of the chord, you'd still have the same notes and therefore the same chord, only that the root note is now an octave higher and no longer an actual root note. For the C major 7th chord above, the 1st inversion of the chord looks like this:



C is put on top of the chord and is an octave higher, whereas E is now the current root note. A 7th note has 4 chords so it has 3 inversions that can be performed. Both the chords and the music scales satisfy the **Closure** group axiom, since even if we perform inversion on the chord or continue our music scale, we still end up having the same elements in the same chord/scale (i.e. group).

Chord inversion can be visualized by assigning each notes of the chord to each cubie's face of the Pocket cube's face, like so:



C major 7th chord is assigned to a cube's face and we read the music notes in clockwise direction. By applying a 90^0 counterclockwise move to the face, we perform the 1st inversion on the chord. Continue to do so and we can create 2nd and 3rd inversions. Ultimately, this is our very final goal of the problem, that is playing the appropriate chords based on the Pocket's cube solving moves. The chord progression won't be played in order so it's exciting to see which chord (or its inversion) is going to be played out.

III. Applying Computational Thinking and Problem – solving Skill

1. Understand the problem
   - What we know?

     We're given the start configuration, which is the scrambled cube, and we know the end configuration which is when the cube's got solved.

     There're $8!\,x3^8$ possible configurations of the Pocket cube. In order to solve it optimally (i.e. finding the shortest sequence of moves that can resolve the scrambled cube), we will need to traverse through these configurations until we get to the end configuration to actually know what that shortest sequence it.

     Standard notation for cube move is F (for clockwise), Fc (for counterclockwise), F2 (180^0); R, Rc, R2;   U, Uc and U2.
   - **Abstraction**:

     Start and end state/configuration of the cube

     List of 24 integer elements representation of a move and of a configuration

Applying a move and its inverse to a configuration (cyclic permutation)

Music key/scale

Build a 7th chord and 2-5-1 chord progression

Chord inversion

- **Decomposition**:

  A module to handle the configuration of the Pocket cube (i.e. store the moves, applying moves to create new configurations)

  A separate module to solve the cube

  A module to get/store the music notes and music scale; create chords, perform chord inversion and assign chord

  A module to play chords based on the input Pocket cube's solved moves

2. Plan a solution

- **Pattern Recognition**: Our approach of solving the Pocket cube is quite similar to solving the Sudoku/Tic-Tac-Toe problem from the lecture "Algorithms and Applications", that is going through the possible configurations of the puzzle to determine the solution/path to winning.
- Strategy:

  For this large project to be modular and easy to debug, we create separate modules to deal with different components of the problem:

  - *rubik.py* - handle the configuration and the moves of the Pocket cube.

As we have discussed previously, we index our cube with integers from 0-23. Thus, we have a 24-element list and our solved configuration of the state will be encoded as:

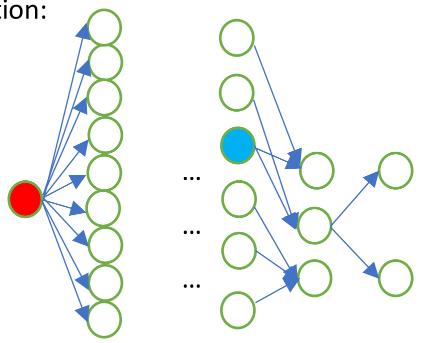
$$(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23)$$

Any other configuration means that the cube is scrambled (or unsolvable). In the end, we want to bring our scrambled cube back to this original configuration (**Identity**).

We also discussed how to represent a move which is part of the cyclic permutation, and we represent it still with a list of 24-element like above. Our task now is to work out the rest of the moves for the faces and hard code them.

Applying a move to a configuration is fairly easy in code once we understand the cyclic permutation. We have an input configuration and a move, and we return a new configuration after applying that move. This can be done with just indexing the original configuration and the move list inside a loop. Getting an inverse move is also straightforward since we do the inverse of what we did when applied a move. We have an input clockwise move to return the corresponding counterclockwise move, and this will still be done with a loop and some indexing of the move list.

A Pocket cube has millions of possible configurations and we cannot afford to store all of them each time we run the program. Instead we only generate them as we go and as we need. This method is used along with the algorithm to solve the cube. We gradually build up a configuration graph where each vertex represents a possible configuration reachable from the previous vertex. Since we have 9 moves/twists that can apply, for each vertex we can generate 9 new configurations reachable in 1 move. We return a dictionary of moves and the new configurations associated with it given an input vertex configuration:



This is only a rough estimation of the representation of the graph. In reality, the graph grows exponentially since each vertex has 9 outgoing edges. Our starting vertex is the solved state and our end vertex (which is our scrambled state) can be anywhere in the graph. The length or depth of the graph is called the diameter, and the Rubik people call this God's Number, meaning what's the greatest number of moves you can make to solve any configuration of the cube. It's proven to be 11 and so the graph above will only have 11 layers in total, but in between there are millions of vertices to traverse through.

- *solve.py* - algorithm to solve the Pocket cube

We want to traverse this Pocket cube's configuration graph from the start vertex configuration until we reach the end vertex configuration. The graph is unweighted, meaning the edges have no weight or there's no cost for moving from 1 vertex to the other. To solve the Pocket cube in least number of moves is to compute the shortest path from the start to end vertex in this type of graph, and we have 2 graph traverse algorithms we can use: Breadth-first Search (BFS) and Depth-first Search (DFS). One of our goal for efficiency is to traverse through this graph as minimum as possible given the exponentially millions of possible vertices. DFS prioritizes the depth of the graph first, meaning from the starting vertex it follows a path, traverses deep into the graph (in this case through 11 layers of the graph) until no more vertex. Then it backtracks and starts again. Our goal is not to traverse the entire graph but just enough of it until we reach our end vertex. DFS traverses unnecessary deep so BFS is the right choice to use whose mechanism is to explore all the neighbor vertices in this current level first before moving to the next vertices at the next level of the graph (it prioritizes and explores the breadth of the graph). Additionally, when BFS reaches the end vertex then the path from the start to end vertex it's guaranteed to be the shortest path.

We cannot afford to store the entire configuration graph so instead we have an Implicit graph representation where we don't know what the next vertices and edges are but can generate them. This is implemented with the help of the function from our previous *rubik.py* module *neighbours().* There are 2 similar approaches to solve the cube, both are BFS but the latter's more efficient than the first:

- Approach 1: pure BFS (1 - way BFS)

BFS explores the breadth of the graph (all the neighbor vertices) first mainly because of the Queue ADT's mechanism "first-in first-out". Initialize a queue storing the vertices that are going to be visited next. We also implement a dictionary to store the parent of the current vertex (i.e. the previous vertex that the current vertex is generated/came from). The parent of start vertex can be either *None* or itself. This will help backtrack to the source vertex and compute the shortest path.

While we still have vertices to visit from the queue, dequeue to get a vertex and get its neighbor vertices. Traverse through these neighbor vertices, if we already visited it (i.e. in the parent dictionary) then skip, otherwise append it to the queue and now the current vertex is the parent of this neighbor vertex. It makes sense that parent dictionary should store the parent of a vertex in terms of configuration and the move to apply, otherwise we don't know how to reach that configuration vertex.

If we visited the end vertex (i.e. it's in the parent dictionary), then break out of the loop and compute the shortest path *get_path()*. Given the parent dictionary as input, we backtrack until the parent of something is *None*, then we know the only vertex like that is the source vertex. The idea is to keep taking the *parent[parent[parent[…[parent[end vertex]]…]]]*, and while doing so we're constructing the shortest path from source to end vertex.

- Approach 2: bidirectional BFS (2 - way BFS)

We can improve a lot by observing the graph and what we know. The graph seems to be directed from 1 end, meaning from the source you can only traverse the graph forward in 1 direction. But actually we can traverse the graph from the opposite direction as well, since we can reach the vertices backward by applying the inverse moves on the configuration (e.g. clockwise forward = counterclockwise backward move). Therefore, the configuration graph is undirected.

Bidirectional search means you search for something from both ends, given that you know the start and the end points as well as be able to access from both directions. We know the graph is undirected and the start and end vertices are given as input, so we can perform a bidirectional BFS on the graph.

The idea is simple as we do BFS on both ends until they visit the same configuration. We immediately stop and backtrack from this common configuration to the 2 ends and add the 2 paths.

- *music.py* - a module to store notes, music scales, construct chords, chord inversion and chord progression

We have a series of .wav files that stores the piano notes (a total of 24 notes spanning over 2 octaves), and using Pygame we create Sound objects that can play these audio files.

We store/hard code all the relevant information in terms of music theory, i.e. the music scales, 2-5-1 chords progression and the 3 inversions (we like 0-base indexing), what cube move corresponds to which inversion, etc. …

Given a root note and a music scale as inputs, we start from that root note and build up the chord using loop and modulo operator to avoid indexing out of the scale. Given a music scale we can also construct the chords in 2-5-1 progression by iterating through that scale to get the corresponding root note 2nd, 5th and 1st and build up the chords from there.

We perform chord inversion exactly how we did when applying a move to a cube's configuration since they share the same idea of cyclic permutation.

- *play.py* - play out the chords given the Pocket cube's solved moves

Prompt user to choose a music scale to play in. Then build the chord progression and assign each chord to each face. From there perform chord inversions corresponding to the moves and saves all the new chords in a new dictionary.

Loop through the moves and look up in the dictionary to get the corresponding chord and play it. User has 2 choices: to play in arpeggio means play the broken chords in separate notes or to play in chords means play the harmonious chords together.

IV. Evaluation

There are 2 main modules that contribute to the goal of the problem, that is solving the Pocket cube *solve.py* and playing the chords *play.py*. The algorithm to play the chords will run in constant time since there're only 24 notes to work with. There are more notes out there spanning over several octaves, but practically there's a limit and musicians only work with 4 octaves at most (48 notes in total).

A much more interesting algorithm to discussed is our 2 versions of BFS algorithm. Given a graph G containing a set of vertices V and edges E, generally the runtime of BFS is O(V + E). For each vertex from the queue, we have to check its edges (i.e. the neighbor vertices). This is equivalent to a linear run time since we may visit some vertex configurations a few time but for each edge we only check once. More theoretically, the runtime is determines as O(max (V, E)) when it comes to asymptotic complexity analysis where either V or E grows faster and dominates the other term. This depends largely on our graph. For a dense (densely connected) graph, meaning every single vertex can connect to each other, E significantly dominants V since $E = \frac{V(V-1)}{2} \Rightarrow O(E) = O(V^2)$ time (the handshaking lemma). Although our configuration graph grows exponentially with millions of vertices, it's actually a sparsely connected graph and therefore $E \approx V$ and the runtime of BFS on this configuration graph is just $O(V)$.

The graph that we implement as implicit graph representation is quite special and we can look at the running time from a different perspective. For a large implicit graph like ours that grows exponentially, the running time of traversing the graph is $O(b^d)$, where b is the branching factor (in our case it is 9) and d is the distance from start to end vertex. In worst case the 1-way BFS will traverse through the entire diameter of the graph, which is 11 levels, and the total runtime for that is $O(9^{11})$ which although is a constant it's very bad.

The bidirectional BFS can do twice the work that 1-way BFS does alone but faster. Since it starts from both ends, each end will only do the work of $O(b^{\frac{d}{2}})$ in worst case where the common vertex configuration is at the middle of the graph. Thus, the time complexity of a 2-way BFS is $O(b^{\frac{d}{2}})$ which is several orders of magnitude more efficient (just compare $9^{11}\ vs.\ 9^{4.5}$). In reality, the common vertex configuration will rarely be in the right middle of the graph. In a Rubik's cube, there are multiple ways of getting to the same specific configuration, therefore at some point both ends will reach the same configuration even though that vertex configurations can be levels apart. This is what makes 2-way BFS significantly better since for the 1-way BFS it struggles to get over that exponential "hump" near the middle of the graph. But generally, 2-way BFS would still run in linear time $O(V)$ asymptotically.

The space complexity in worst case is $O(V + E) = O(V)$, or $O(b^d)$ for 1-way BFS since we're storing the vertices needed to visit next in the queue. For 2-way BFS, we can improve the memory complexity as $O(b^{\frac{d}{2}})$, since we only need to visit these numbers of vertices in worst case and store them in the queue.

Charles Nguyen

V. Conclusion

Overall, the project has covered a lot of material used in class as well as outside topics. Queue ADT implementation is one of those, along with the idea of solving a puzzle by representing it as a graph in lecture “Algorithms and Applications”. Dictionary is used almost always for its O(1) look-up time especially in a graph of millions of vertices. There’s no recursion but the project uses a bit recursive thinking when we’re backtracking to the source vertex in BFS. Additionally, the basic axioms and cyclic permutation in group theory helps us working with the cube more discretely, and graph analysis is used to evaluate the efficiency of the 2 BFS algorithm.

I think the project has been a great success. A major part is the algorithm design to solve the cube and I was able to implement not just 1 but 2 versions of BFS and analyze the huge difference between them in terms of runtime complexity although they are essentially the same BFS algorithm. I was able to observe the graph carefully and design a significantly better algorithm, and thus not only I achieve the goal of solving the cube in less than 2s but I can do it in less than 1s!

Another major, very interesting part of the project is playing out the chords/melody pleasantly and the project definitely achieved that goal. It is the one part that I doubt most it will work, but in the end it turns out to be better than I expected. The entire process has been really difficult as I’ve gone through many topics related to music theory and group theory and actually discovers a lot of new things that I can implement to finally have this successful project. Through the project and the analysis it is clear that the 2 almost distinct fields of music theory and group theory is actually very much related to each other in the most surprising way, and I hope the general audience can appreciate this connection.

Extension:

- The project was very close to be more intuitive where the cube can be displayed using SimpleGraphics module. I even have a function in rubik.py module to start assigning colors to the 24 cubie’s faces. The only next step is to apply the functions *rect(x, y, w, h)* and perhaps *setFill(), setOutline()* or *setColor()* to display the colors.
- The bidirectional BFS is sadly not perfect. It does not guarantee the shortest path (the path it constructs is usually just 1 solved move longer than the 1-way BFS’s). The next step is to research on this and come up with a fix on this problem.
- There are modules such as *rubik.py* and *music.py* that was trying to make the other main modules solving the actual problem look cleaner. These modules’ only task is to create abstraction so the code quality is not the best with a lot of hard coding. Although they’re not the main parts of the project it never hurts to make the module’s codes cleaner.