

God's Algorithm in The 2x2x2 Rubik's Cube

Muhammad Mirza Fathan Al Arsyad - 13518111

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

13518111@std.stei.itb.ac.id

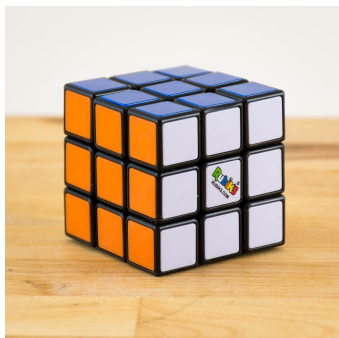
Abstract—The Fabulous Rubik's Cube. A simple idea yet addicting. People all over the world are trying to solve the puzzle toy. And then, solving is not enough for them. They started something that we call the speedcubing, an act of solving the Rubik's Cube in the fastest way possible. According to the World Cube Association (WCA), the current record time for the 3x3x3 sized cube is in unbelievably 3.47 seconds! The 2x2x2 sized category is crazier, a Polish Cuber solved it in just 0.49 seconds, which isn't even a half second! How could that be possible?

Keywords—Cube, DFS, Graph, Permutation.

I. INTRODUCTION

The Rubik's Cube (will be referred as 'Cube') was invented by a Hungarian Architecture Professor, Erno Rubik in 1974. Originally it is literally a cube-shaped toy, with 6 faces having their own colour stickers. The original size 3x3x3 means that it is 3 cubes in length, width, and height. The same thing is also applied on the 2x2x2, 4x4x4, 5x5x5, and all nxn sized Rubik. We can conclude that each face (which is square shaped) has 3x3 cubes or 9 cubes, meaning these 9 little cubes are covered by 9 stickers with the same colour. If that's the case, then the Cube is on the solved state. Else, it is the unsolved state.

The Cube consists of rotatable layers, as shown in the picture below. These rotatable layers make it possible for the cube to have more than one colour in one face, which is the unsolved state. This is why people are getting addicted to this little toy. Finding a way to put the Cube in a solved state is challenging yet fun, for those who love challenge. Once you succeed, you are a solver.



The Solved State of an Original (3x3x3) Rubik's Cube

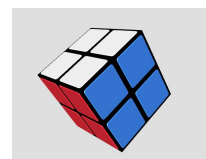
Source: <https://www.menkind.co.uk/cube>

People are trying to be more than just a solver, once they solve the Cube. They want to be fast. They compete with each other.

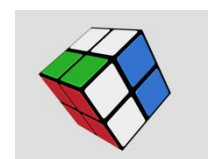
The Rubik's Cube are basically consist of states. One solved state and the other are the unsolved states. And this states can be stored inside a data structure called graph. This paper is going to show how the 2x2x2 sized Rubik's Cube can be implemented in a graph.

Why graph? Because each state can be obtained from the other state, by rotating the layers of the Rubik's Cube, and it shows how do those graph elements (the Cube's states) connected to each other. For example, in the picture below, picture (b) state can be obtained by rotating one layer from the picture (a) state (which is the solved state).

And then, we can find the shortest path between two nodes (graph elements) with a traversal algorithm of graph, which is going to be the Depth First Traversal.



(a)



(b)

Figure 2. Example of state transition from (a) to (b)

Source: Online 2x2x2 Rubik Simulator

<https://www.grubiks.com/puzzles/rubiks-mini-cube-2x2x2/>

II. BACKGROUND ON GRAPH

A. Definition

A graph $G = (V, E)$ consists of V , a nonempty set of vertices (or nodes) and E , a set of edges. Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints. (From Reference[1]). Simple graph, is a graph with no loop or parallel edges connecting a pair of vertices.

B. Types Of Simple Graph

1. Complete Graphs

A complete graph on n vertices, denoted by K_n , is a simple graph that contains exactly one edge between each pair of distinct vertices. If there's any pair of distinct vertex not connected by an edge is called noncomplete.

2. Cycle

A cycle C_n , $n \geq 3$, consists of n vertices v_1, v_2, \dots, v_n and edges $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}$, and $\{v_n, v_1\}$ [1]. It's like a circle.

3. Tree

A Graph with no Circuit

4. Regular Graph

A Graph which consist of all vertices having the same number of degrees. Degree of a vertex is the number of vertices that is adjacent to a vertex.

And there are still many more but i think it would be unnecessary to discuss every type of graph, because in the Rubik's Cube case we are going to need only regular graphs because every vertex has the same degree.

C. Graph Representation

It's easy for human to represent graph as like what we see in pictures, using circles as nodes and lines as edges. As computer store data using the binary numbers, we cannot easily draw a graph in a computer We need another way that is possible for the computer to 'understand' the graph we are talking about. There are many methods, but the most well known is the adjacency list and the adjacency matrix.

III. BACKGROUND ON THE RUBIK'S CUBE AND THEIR ALGORITHMS

A. Possible Permutations

In the $2 \times 2 \times 2$ Cube, there are 8 pieces of cubes. Therefore, we can conclude that the number of possible permutations for this kind of Cube is the factorial of eight or usually denoted as $8!$.

But we're not done yet. Each piece of cube has 3 possible orientations, as shown in the figure below.

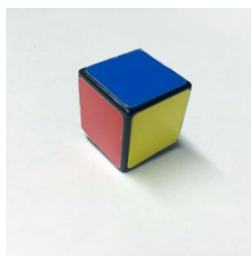


Figure 3. Each piece of $2 \times 2 \times 2$ Rubik's Cube has three faces therefore, it has 3 possible orientations.

Source : solvemrubikscube.com/images/corner-2300.png

Since those 8 pieces have 3 possible orientations, the total possibilities can be calculated as 3 to the power of 8.

Therefore, combined with the $8!$,

$$P(C) = 8! \cdot 3^8$$

Where $P(C)$ is the number of possible permutations of the Cube. The number of $P(C)$ would be 264539520 which is not a bad number for computational problems. We're going to represent these permutations as the graph nodes.

B. Cube Algorithms

There are many possible ways to solve this Rubik's Cube Puzzle. Cubers called these 'ways' as 'algorithm', not to be confused with the algorithm in the programming context. Fundamentally, both of them (both the algorithm in the cube's context and the programming one) are a series of steps that are needed in problem solving.

The algorithm has several conventions in writing their notations. Each notation has their own meaning on what to be done to the cube. Look at the figure below.

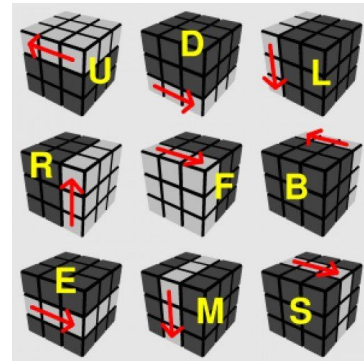


Figure 3. $3 \times 3 \times 3$ Cube algorithms

Source : https://usercontent2.hubstatic.com/7997215_f520.jpg

The figure visualizes how does the algorithm work on a $3 \times 3 \times 3$ Cube. Those U, D, L, R, F, B, E, M, S stand for Up, Down, Left, Right, Front, Back, Equator, Medium, and Slice respectively, which are the layer that we're rotating. In the $2 \times 2 \times 2$ Cube that we're discussing, it has nearly no difference. The only difference is that in the $2 \times 2 \times 2$ Cube we don't have the E, M, and S algorithm because we have no central layers in the $2 \times 2 \times 2$ Rubik's Cube. It makes our task to solve the cube a lot simpler than the $3 \times 3 \times 3$ Cubes.

If you look at the figure above carefully, you would realize one thing, that all U, D, L, R, F, E, M, S, B are turning the Cube layer clockwise. How about the notations for the anti-clockwise turns? It will be $U', D', L', R', F',$ and B' . There's even a notation about rotating a layer twice, which are $U2, D2, L2, R2, F2,$ and $B2$. We don't need to make a convention for rotating a layer three times clockwise because it will definitely be the same as rotating them in the anti-clockwise way, and vice versa.

These letters (U, L', B2, and so on) is going to be our edge in the graph which connect each state of the Cube's condition.

C. How It Relate to Graph Problem

As already mentioned above, the node of the graph will represent each possible state of the cube, which is the number $P(C)$. It is a regular graph since it's nodes are all having the same degree, which is 18. The number 18 obtained by counting the possible move that can be done to the cube, which are the U, U' , $U2$, R, R' , $R2$, L, L' , $L2$, D, D' , $D2$, F, F' , $F2$, B, $B2$, and B' . These edges are connecting every possible state.

Our task is to find the most optimal solution, which is the ‘God’s Algorithm’ for each possible case in solving the 2x2x2 cube. Because the fewer steps you made in solving the cube, the faster you are going to solve it.

IV. DEPTH-FIRST SEARCH

A. Definitions

Depth-first search (DFS) is an algorithm to traverse, or to search on a graph data structure. The algorithm starts on the node called a root, and explore the graph far as possible through the edges, before backtracking.

B. Pseudocode and How it Works

In the following graph, as an example, we start the search/traversal from the vertex 2. When we arrive in the vertex 0, we have to find all adjacent vertices/neighbouring vertices. Then check if it is visited or not. If it isn’t then we visit that vertex. If it is, then we start to do the backtracking. A DFS for the graph on the figure below will be 2, 0, 1, 3.

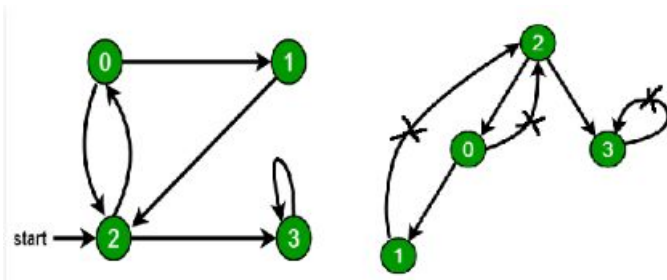


Figure 4. Depth First Search

Source:

[geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)

The pseudocode below is explaining how does the DFS work briefly.

```

procedure DFS( $G, v$ ):
    vertex  $v$  is visited
    for all nodes from  $v$  to  $w$  that are in
     $G.\text{neighbor}(v)$  do
        if vertex  $w$  is not visited then
            DFS( $G, w$ )
    
```

C. Solving the Cube with DFS

We are going to need $8! \cdot 3^8$ vertex to represent every state of the Cube. Even though it is possible for the computer to store all of them, it would be a waste of time if we have to generate each state. We aren’t going to have the vertex of the graph before implementing the DFS algorithm like the usual graph problems due to that reason.

We are going to generate the graph with their node while doing the search algorithm. It’s actually a bit different than the pure DFS algorithm, but fundamentally, we implement the concept of traversing a graph as deep as possible until either we find the node that we are looking for or we are going too far so that we have to stop the traversal because if it is too far

that it shouldn’t be the God’s Algorithm. We can limit the number of vertices to be traversed to a certain number, let’s say 100, to prevent the possibilities of infinite recursions. (Let’s say if we are traversing through a cyclic graph). The number 100 is reasonable and should be considered high enough because first, even the most simple Cube algorithms such as the LBL can solve the cube below that number of steps, second, it’s the ‘God’s Algorithm’ so that if it takes 100 steps it shouldn’t be it, third, if it’s possible for a human to put it into a solved state in under a half second (a Polish Cuber Maciej Czapiewski solved it in just 0.49 seconds, current record holder for the 2x2x2 Cube), then how it could be possible to make 100 moves in such a little amount of time?

Then if we are about to implement the concept of DFS to traverse the vertices of Cube states, then doesn’t DFS have a starting node called a root? Would we need that?

The answer is yes. The root vertices will be the solved state. After assigning the solved state to the root of the graph structure, then we start traversing all the states by doing some rotations to the Cube such as R, L2, and so on. As mentioned before, we are going to stop the traversal until we find the node that we are looking for. What node is it? It’s the current Cube state (the current situation that we are going to solve). After we find the cube state that we are facing then the traversal has come to its end.

Don’t forget that we have to store the number of vertices we traverse so that the program knows how far it is from the beginning vertex the solved state. If it arrives at the node that we are looking for, then store the number of steps in an array. If it doesn’t find the destination vertex until it comes to the hundredth step, then store the 100 to the array. We will need at least 18 for the array size since we have 18 kinds of slice rotations (U, U2, U’, R, R2, and so on). Dynamic array would be preferred since we are going to need many allocations so that we can store it to the heap memory.

After that, we still have to traverse the vertices through another path (edges or the Cube layer rotations). After the array are all filled, then we will return the minimum value of the array, which is the shortest path. Each step in finding the current state from the starting solved state will also be memorized in an array so that we know how can we solve the cube in an efficient algorithm.

We will add another stop condition, which will happen if the traversal arrives to the starting solved state to optimize the program a bit.

We are solving the cube backward, by starting on something that should be the destination. There’s also another difference from the original DFS algorithm. The DFS won’t visit a node that it has already visited in the previous traversal. It can’t be the case when we are solving a Rubik’s cube. DFS algorithm’s main purpose is just to traverse all the element while we are trying to find a vertice, and it’s a different thing.

D. Complexity Analysis

Since there are 18 choices of neighboring vertices from the root, and suppose the number of traversals is N , then there will be 18^N traversal, which can be denoted the $O(2^N)$ in the big-O notations. This is another difference of this algorithm to solve the 2x2x2 Cube and the original DFS, which has $O(|V|)$ as it's complexity value in the big-O notation, where $|V|$ is the number of vertices in the graph data structure. We are actually preventing that from happening because in our case, we will have $8! \cdot 3^8$ as the number of vertices and it is not a small number, and seems to be a waste of memory.

V. IMPLEMENTATIONS IN C++ PROGRAM

In the C++, I construct an abstract data type called the RUBIK, which is structed from another abstract data type called SIDE. The RUBIK represent the current condition of the cube, and SIDE represent the face of the Cube. For the color, I made another program to modify the I/O interface colours so that it can show colours even if it's only a command line based program.

a. The SIDE Data Type

```
typedef struct {
    int TabSide[2][2];
    int Left[2], Right[2], Up[2],
    Down[2];
} SIDE;
```

Figure 5. SIDE Data Type
Source: Author

The SIDE data type is constructed from a 2 dimensional array and some arrays. The TabSide is a two dimensional array of integers to store the data of what colour the SIDE is. The Left, Right, Up, and Down is representing the layers, that can be twisted/rotated. Some selectors would also be needed to make a more readable code.

```
#define Elmt(S,i,j) (S).TabSide[i][j]
#define Left(S) (S).Left
#define Right(S) (S).Right
#define Up(S) (S).Up
#define Down(S) (S).Down
```

Figure 6. SIDE Data Type Selectors
Source: Author

b. The RUBIK Data Type

```
typedef struct {
    SIDE TOP;
    SIDE DOWN;
    SIDE RIGHT;
    SIDE LEFT;
    SIDE FRONT;
    SIDE BACK;
} RUBIK;
```

Figure 7. The RUBIK data structure.

Source: Author

The RUBIK data type is constructed from 6 SIDEs, which represent the 6 faces of a Rubik's Cube (surely every cube has 6 faces). The SIDE name corresponds to the face it's representing, for example, SIDE DOWN is the down face/layer of the Rubik's cube, the SIDE LEFT is the left hand face/layer of the Rubik's cube, and so on. We would need to implement selectors too, to make our code readable.

```
#define TOP(R) (R).TOP
#define DOWN(R) (R).DOWN
#define RIGHT(R) (R).RIGHT
#define LEFT(R) (R).LEFT
#define FRONT(R) (R).FRONT
#define BACK(R) (R).BACK
```

Figure 8. The RUBIK Selectors

Source : Author

c. Movement Algorithms

Movement such as rotations to the layers need to be implemented to, like the R, or L, and so on. The figure below shows an R algorithm

```
void R(RUBIK *Rb) {
    int temp[2] = {Elmt(TOP(*Rb),0,1),
    Elmt(TOP(*Rb),1,1)}; // top
    Elmt(TOP(*Rb),0,1) =
    Elmt(FRONT(*Rb),0,1);
    Elmt(TOP(*Rb),1,1) =
    Elmt(FRONT(*Rb),1,1);
    Elmt(FRONT(*Rb),0,1) =
    Elmt(DOWN(*Rb),0,1);
    Elmt(FRONT(*Rb),1,1) =
    Elmt(DOWN(*Rb),1,1);
    Elmt(DOWN(*Rb),0,1) =
    Elmt(BACK(*Rb),0,1);
    Elmt(DOWN(*Rb),1,1) =
    Elmt(BACK(*Rb),1,1);
    Elmt(BACK(*Rb),0,1) = temp[0];
    Elmt(BACK(*Rb),1,1) = temp[1];

    int temp2 = Elmt(RIGHT(*Rb),0,0);
    Elmt(RIGHT(*Rb),0,0) =
    Elmt(RIGHT(*Rb),1,0);
    Elmt(RIGHT(*Rb),1,0) =
    Elmt(RIGHT(*Rb),1,1);
    Elmt(RIGHT(*Rb),1,1) =
    Elmt(RIGHT(*Rb),0,1);
    Elmt(RIGHT(*Rb),0,1) = temp2;
}
```

Figure 9 . The R procedure, or twisting the Right layer of the cube in a clockwise direction.

Source: Author

Fundamentally, it is no more than a series of swapping procedures. At first, the right layer from the top side will be stored in a temporary variable, and then the right layer of the top side will be overwritten with the right layer elements in the front side. The front side will be overwritten from the down side, the down side from the back side and the values that were previously stored in the temporary variable will be copied and overwrite the back side right layer elements.

After that, we have to rotate the right face too, by swapping it's elements in a clockwise variable, by using the help of a temporary variable as well.

The R' algorithm, is equivalent to the R repeated three times, and I implemented it only by repeating the R procedure three times, as the figure shown below.

```
void r(RUBIK *Rb) {
    /* The R' Algorithm */
    R(Rb);
    R(Rb);
    R(Rb);
}
```

Figure 10. The r procedure, which is turning the right layer in an anti-clockwise direction, and it's equivalent to repeating the procedure R three times.

Source: Author

The R2 works in the same way, by just repeating the R twice. It would be inefficient as the program would do swapping assignment more than once. It would be more efficient if we just swap it like how the R procedure works, but this time, I am just trying to make the program shorter and easier to debug. The other algorithms, L, U, D, and so on, works in a similar manner, so it wouldn't be necessary to show the implementation in this paper.

d. Solving

And finally, the solving algorithm.

```
RUBIK Search;
RUBIK Problem;

int Solve(RUBIK *Search) {
    vector<int> count

    if (IsSameState(Problem, *Search)) {
        /* State found! */
    }
    if !(IsSameState(Problem, *Search)) {
        R(Search);
        Solve(Search);
        count[0]++;
        r(Search); /* Back to the
previous state */
    }
}
```

```
r(Search);
Solve(Search);
count[1]++;
R(Search);

/* .. The full code is at
github.com/mirzaalarsyad/Pocket-Cube-Solver
*/
}
return MinimumArray(count);
}
```

Figure 11 . The Rubik Solving Procedures.

Source: Author

The section IV.d. already explained a lot about the algorithm. But as an additional brief explanations, the Search RUBIK global variable is the RUBIK that we are going to use as a comparison to the Problem, another RUBIK global variable while we traverse the graph. The Search rubik is going to be twisted and compared, and when the IsSameState(Problem, *Search) returns a true value, it means that the Search RUBIK is the same with the Problem RUBIK and it means that the Cube is solved. IsSameState(RUBIK Rb1, RUBIK Rb2) is nothing more than an additional function that I've defined in another section of the program which returns true if and only if Rb1 equals to the Rb2. The Solve will return the minimum value of dynamic array count. MinimumArray is another defined function that returns an integer value, the minimum size of an array.

After several test cases, the program seems to return various values, all below 20. Mostly between 15 to 19, sometimes under 10. Which is considered to be the number of steps required to finish the current state. The true 'God's Number' for the 2x2x2 Rubik's Cube is actually 11, based on the World Cube Association. If it's more than 11, then it's inefficient, like what I got in the test cases. This could be because there are some other Cube Algorithm which isn't implemented, like the whole cube rotation (rotating the Cube's orientation). And the IsSameState(RUBIK Rb1, RUBIK Rb2) returns false even if Rb1 and Rb2 are actually the same state, but in different orientation. Let's say, two solved Cubes are considered different if one cube has the yellow face in the top side, while the other has the red colour.

VI. CONCLUSION

The DFS is just an alternative to find the God's Algorithm to the Rubik's Cube. While the number 11 is considered as the number of maximum steps needed to solve a 2x2x2 Rubik's Cube, it makes sense if world class Cubers can solve in just 1 second or below.

VII. ACKNOWLEDGMENT

First, the author would like to thank God for giving the author the ability and chance to finish the paper. The author would also like to thank Ms. Fariska Zakhralativa, S.T., M.T., as the lecturer of Discrete Mathematics for the guidance given for this semester. In addition, the author would also thanks to author's parents, family, and everyone who supports the author in finishing the paper.


REFERENCES

- [1] K. H. Rosen, Discrete Mathematics and Its Applications, 7th Edition, 2013
- [2] World Cube Association, <https://www.worldcubeassociation.org/> accessed December 6, 2019

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 6 Desember 2019

A handwritten signature in blue ink, appearing to be 'M. F. Al Arsyad', written on a light-colored background.

Muhammad Mirza Fathan Al Arsyad - 13518111