Rubik's Cube Solver

Adittya Gupta

Introduction

The aim of this project is to implement the **Herbert Koceimba** algorithm using C++. The algorithm was discovered in 1992 by Koceimba himself and is said to be an improved version of the Thistlethwaite's algorithm. The algorithm claims to find the solution to any solvable cube orientation in less than 30 moves. Although there haven't been any instance of a cube that took more than 20 moves(The God's number) after so much testing.

The implementation here is quite basic and straight-forward and doesn't implement all the features of the original Koceimba algorithm because the original Koceimba algorithm although promising the upper bound on the number of moves, had a very high running time even on a high-end device. The aim of this project is to just get a nearly optimal solution in least amount of time possible.

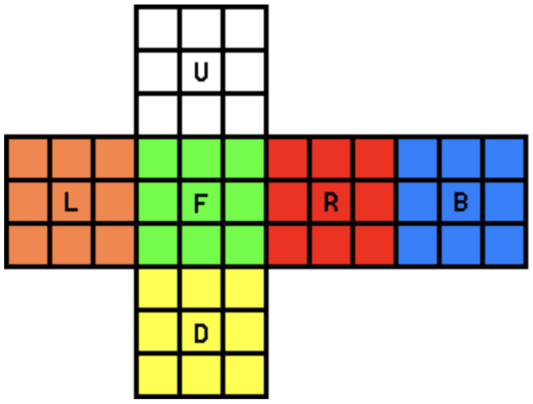
The original and full implementation of this algorithm is done by Koceimba himself and is present on his website(You can find that website here). The implementation was done in Pascal. Moreover, most of the implementation of this algorithm that I could find on the internet was in either Python or were not fully correct. This motivated me to implement this in C++ because of the speed advantages that it offer over Python and the ease of readability and modernity that it offers over Koceimba's Pascal implementation.

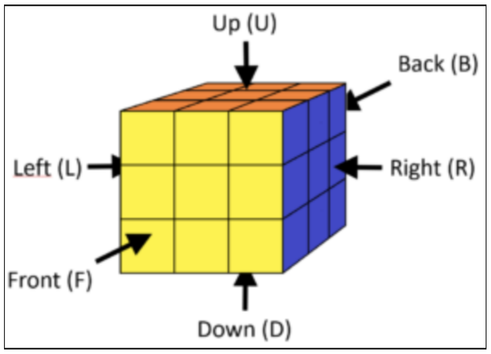
The Two Phase Algorithm

This algorithm is also known as the Two Phase algorithm. The reason being it does the solution searching in two phases. Consider a solved cube and suppose I allow all the moves except {R, R', L, L', F, F', B, B'}. Then, it appears that we get a subset of the original set of all the solvable cube instances. The property of this subset is :-

- The orientation of all the corners in this subset remain same as it was before even after applying the allowed moves
- The orientation of all the edges in this subset remain same as it was before even after applying the allowed moves

- The four edges in the UD-slice (between the U-face and D-face) stay isolated in that slice.





We can look the set of all solvable cube instances as a group on permutations G. Suppose if I am given any cube instance then if I write it in this form

$$U_1U_2U_3U_4U_5U_6U_7U_8U_9R_1\dots R_9D_1\dots D_9L_1\dots L_9F_1\dots F_9B_1\dots B_9$$

Then I can define an operation $*$ on this group as the product of two permutations or product of two cubes with the identity element being the solved cube.

The benefit of this is that now all the moves to the cube C can be seen as a $C * M$ where M is the cube obtained by applying that move on the solved cube.Now let H denote the subgroup of G which we were talking about earlier i.e. generated by restricting some type of moves.

So our algorithm will solve the cube in two phases first it will find a sequence of moves that will bring the scrambled cube to the subgroup H and then we will find a sequence of moves(restricted to this group) to go back to the solved state. Our algorithm in order to do so will use Iterative Deepening using lower bound heuristics (IDA* algorithm) two times back to back. The heuristics that will be used will be defined by coordinates of the current state. I will discuss about it in next section.

We have used IDA* because simple DFS/BFS or simple A* algorithm won't work here because the state space is very large and IDA* doesn't store the visited nodes unlike A* and doesn't go to non-optimal nodes unlike simple DFS/BFS.

Enumeration of all the variables used

Enumerating all the faces of the cube

U	R	F	D	L	B
0	1	2	3	4	5

The above order is used whenever the faces of cube are talked about. Similarly we can enumerate the edges and the corners of the cube.

The edge UF denote the edge containing the Up and Front faces while the corner URF denote the corner containing the Up, Right and Front faces or the corner at top right on the front face

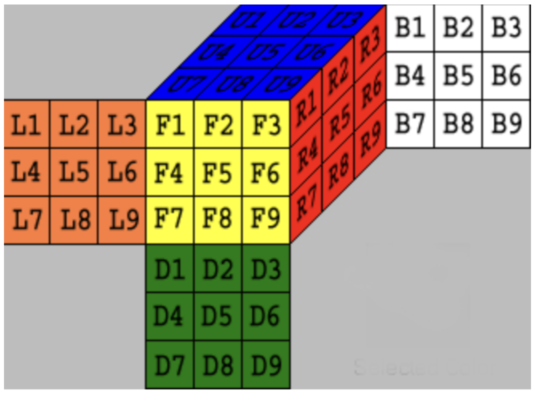
Enumeration of corners is as follows

URF	UFL	ULB	UBR	DFR	DLF	DBL	DRB
0	1	2	3	4	5	6	7

Enumeration of edges is as follows

UR	UF	UL	UB	DR	DF	DL	DB	FR	FL	BL	BR
0	1	2	3	4	5	6	7	8	9	10	11

In a similar way we can label each facelet of the cube starting from U_1 till B_9 in the order which was given in the last page we can label them from 0...53.



Coordinates

The information about a particular state in a phase of the algorithm can be determined quickly by defining some coordinates. All the cubes that have same coordinates will have identical cost to get to the solved state. For example, If I apply F-move to a solved cube and U-move to another solved cube although both cubes are very different in their structure but both of them could be bring back to solved state by only applying one move so we would ideally want both of them to have the same coordinate

Phase I coordinates

The coordinates that define any cube instance during the first phase of the algorithm are given below. The cube can be declared in the H subgroup if all the below coordinates are 0 and this is easy to follow once you clearly understand the definition and significance of these coordinates.

- **Twist**

  It denotes the twist(or the orientation) of the 8 corners of the cube. Take any corner of a cube it can be in only three orientations denoted by 0, 1 or 2 where 0 represents that it is in the same orientation as it would be in the solved state. In the implementation the twist coordinate is specified by a number between $0 \ldots (3^7 - 1)$ signifying there are three possible orientations for each of the 8 corners so we get a ternary number represented in decimal. This number is zero when twist of all the corners is 0 in the cube

- **Flip**

  It denotes the flip(or the orientation) of the 12 edges of the cube. Take any edge of a cube it can be in only two orientations denoted by 0, 1 where 0 represents that it is in the same orientation as it would be in the solved state. In the implementation the flip coordinate is specified by a number between $0 \ldots (2^{11} - 1)$ signifying there are two possible orientations for each of the 12 corners so we get a binary number represented in decimal. This number is zero when flip of all the edges is 0 in the cube

- **FRtoBR**

  It denotes the number position(not permutation) of the edges between the U and D face in the cube. When it's 0 it means that all the four edges between the U and D face are present in the UD slice. It is a number between $0 \ldots {}^{12}P_4 - 1)$ since we want the permutation of the 4 edges to remain there only. The way of deriving it for some cube instance is explained beautifully by Koceimba in his website and I omit explaining it here.

Phase II coordinates

Once we get a cube in subgroup H most of the work is done because now the cube only consists of permutation of the edges, corners and UD slice edges. So in order to get to the solved state we need to ensure that all these coordinates have the value 0.

- **URFtoDLF**

  It denotes the permuation of the 8 corners of the cube from URF to DLF. In total there are 8! such permutations so this is a number ranging from $0 \ldots (8! - 1)$. If a cube has this coordinate as 0 then all its corners are in their right place. Again I will not show how it is derived from a given instance of the cube as it is given here

- **URtoDF**

  It denotes the permutation of 8 edges (All the edges except the 4 edges in the UD slice) and is denoted by a number between $0 \ldots {}^8P_6 - 1)$ because it is easy to determine the permutation of the rest two edges if we know the permutation of the rest 6 edges by using parity. You can again look how it is derived from the link already given.

  Another important thing is that in the implementation this coordinate is further divided into two more coordinates because it is extensively used and it may become time consuming calling it again and again. The two coordinates are URtoUL denoting the permutation of three edges {UR, UF, UL} and UBtoDF denoting the permutation of three edges {UB, DR,DF}.

- **Parity**

  It denotes the parity of the corner and edge permutation of the cube instance. It turns out that the parity of the cube is always the same as that of the parity of edges and if it is not so then the cube is unsolvable. It is 0 for the cube in solved state. This is how to calculate the corner and edge parity. Find all the inversions in the corner permutation where ordering is defined in the same way as these corners were enumerated. Now the inversions modulo 2 is the parity of the corner permutation. Similarly we can find the parity of the edge permutation

The Facelet level

The facelet level is denoted similarly to the permutation definition of the group G that we have already given. Given two cubes on the facelet level we can get another cube by following the same operation that we use when we multiply two permutation. For example if P_1 and P_2 are two permutation then

$$P_3[i] = (P_1 * P_2)[i] = P_1[P_2[i]]$$

It is easy to see that this group is not Abelian so we have to be careful in the notation and implementation. The facelet level/permutation of a solved cube is identity permutation. Not much can be done in this level as far as implementation is concerned because although having very easy implementation of applying moves to the cube we can't apply the search algorithm on this because it is very difficult to find the coordinates of the state from its facelet level. We want something more closer to the coordinates and for this we use the CubieCube and the CoordCube level

The CubieCube level

In this level we describe the position of the corner and edges by using 4 arrays denoting corner permutations, corner orientations, edge permutations and edge orientations. I will use the word cubie to imply edge or corner both. The way how we store it is easy to see for permutation just store the permutation in the array and for orientation $or[i]$ denotes the orientation of the cube or edge whatever is applicable.

The important thing about this level or this way of representation is that now we don't define the product of two cubes but we define the product of two moves. For example let $A[x]$ denote the effect of move A on cubie x and $A[x].c$ denote the position in the permutation of cubie x after the move A and $A[x].o$ denote the orientation. Then the following rules hold.

$$\begin{aligned}(A * B)[x].c &= A[B[x].c].c \\ (A * B)[x].o &= A[B[x].c].o + B[x].o \quad (modulo\ 3)\end{aligned}$$

Using the above two rules we can find the orientation and permutation for any sequence of moves. This level is far more better than the facelet level in terms of finding the coordinates although finding the effect of moves on a cube has become a little bit messy. Still a problem is there since this level involves array we have to again and again shift, rotate and change the array which is time consuming and space consuming. To make it more faster we will use the CoordCube level. We will be interchanging between these two levels rapidly in the course of our search to use the one which is best suited for that moment of time

The CoordCube level

Finally the Coordcube level is very similar to how we define the coordinates. We define a cube by it's 8 coordinates that we defined earlier each being a decimal number. We won't apply move directly in this level. If we want to apply a move at some instace of cube at this level we first convert it into the cubiecube level and then apply the operation and then get back to the coordcube level. It is still fine because we are storing a very little space for each cube and inter-conversion between these two levels is pretty fast.

Move Tables and Prune tables

As was mentioned earlier given a coordcube instance if we have to apply some move to it we have to first convert it into a cubiecube instance followed by applying the move on this cubiecube instance and then converting back to coordcube instance. In short it is time consuming so what we can do to improve it is store these values in a table for each coordinate so that if I apply a move on a given coordinate what coordinate I will get to I will pre-compute it before the searching starts.

But we have to pre-compute it anyways so what is the benefit of doing this? The thing is we will store this table locally in a binary file so whenever we need this table instead of computing it we will just load this file. So the first time algorithm will take some time but from the next run it will become very fast because it has to simply load data now and not compute anything

The same can be done for the prune table heuristics. The prune table stores the distance for the current tuple of coordinates to the solved state. It is obtained by applying moves to the solved state of the cube and storing the values obtained there in a table. We can again precompute this table since it will not change again and again.

Prune tables consits of four tables each is a tuple of two-three coordinates Slice_URtoDF_Parity_Prun is a tuple of (UDSlice, URtoDF, Parity) coordinates and is used in phase 2 of algorithm.Slice_URtoDF_Parity_Prun is a duple of (UDSlice,URtoDF,Parity) coordinates and is used in phase 2. Slice_Twist_Prun is a tuple of (UDSlice, Twist) coordinates and is used in phase 1. Slice_Flip_Prun is a tuple of (UDSlice, Flip) coordinates and is used in phase 1

Implementation Details

The project consists of five main files namely FaceCube.cpp, CubieCube.cpp, CoordCube.cpp, IOhandler.cpp, Search.cpp and rest some header files used for definitions. I will show the header files for the above five files to give an idea what is in each class.

FaceCube.h

```
class FaceCube
{
public:
    vector<int> f;
    FaceCube(string cubeString =
        "UUUUUUUUURRRRRRRRRFFFFFFFFFDDDDDDDDDLLLLLLLLLBBBBBBBBB");
    string to_string();
    CubieCube toCubieCube();
};
```

The class member *f* is used to store the permutation of the facelets as was discussed in the section. In addition to that it consists of a constructor and a function to convert f back into string format. Also a function is present to convert it to CubieCube level.

CubieCube.h

```
class CubieCube{
public:
    vector<int> cp;
    vector<int> co;
    vector<int> ep;
    vector<int> eo;
    CubieCube(
        vector<int> cp = {URF, UFL, ULB, UBR, DFR, DLF, DBL, DRB},
        vector<int> co = {0, 0, 0, 0, 0, 0, 0, 0},
        vector<int> ep = {UR, UF, UL, UB, DR, DF, DL, DB, FR, FL,
                          BL, BR},
        vector<int> eo = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0});
    void cornerMultiply(CubieCube other);
    void edgeMultiply(CubieCube other);
    int cornerParity();
    int edgeParity();
    int getTwist();
    int getFlip();
    int getFRtoBR();
    int getURFtoDLF();
    int getURtoDF();
    int getURtoUL();
    int getUBtoDF();
    int getURFtoDLB();
    int getURtoBR();
    void setTwist(int twist);
    void setFlip(int flip);
    void setFRtoBR(int idx);
    void setURFtoDLF(int idx);
    void setURtoDF(int idx);
    void setURtoUL(int idx);
    void setUBtoDF(int idx);
    void setURFtoDLB(int idx);
    void setURtoBR(int idx);
    int verify();
};
```

This class contains the four arrays as mentioned in the section with a constructor with some default values. With this we have a corner multiply and edge multiply functions which do the same thing as was specified by the formulas. After that there is a function to get the parity of the corners and edges which will be used in subsequent functions. After that there are getters and setters for the 8 coordinates that were already specified. After that there is a verify function to check if the cube is even solvable or not. It checks this using basic checks and checking if the corner parity matches with edge parity.

CoordCube.h

```
class CoordCube
{
public:
    int twist;
    int flip;
    int parity;
    int FRtoBR;
    int URFtoDLF;
    int URtoUL;
    int UBtoDF;
    int URtoDF;
    static vector<vector<int>> twistMove;
    static vector<vector<int>> flipMove;
    static vector<vector<int>> parityMove;
    static vector<vector<int>> FRtoBR_Move;
    static vector<vector<int>> URFtoDLF_Move;
    static vector<vector<int>> URtoDF_Move;
    static vector<vector<int>> URtoUL_Move;
    static vector<vector<int>> UBtoDF_Move;
    static vector<vector<int>> MergeURtoULandUBtoDF;
    static vector<int> Slice_URFtoDLF_Parity_Prun;
    static vector<int> Slice_URtoDF_Parity_Prun;
    static vector<int> Slice_Twist_Prun;
    static vector<int> Slice_Flip_Prun;


    CoordCube(CubieCube c);
    void move(int m);
    void setTwistMove();
    void setFlipMove();
    void setFRtoBRMove();
    void setURFtoDLFMove();
    void setURtoDFMove();
    void setURtoULMove();
    void setUBtoDF();
    int getURtoDF(int idx1, int idx2);
    void setMergeURtoULandUBtoDFMove();
    void setPruning(vector<int> &table, int index, int value);
    void makePruningTables();
};
```

This class contains the 8 coordinates required to define the CoordCube level. In addition to this it contains MoveTables and PruneTables which are made static so these need to be initialised only once and then can be used by all the objects.

After that there is a constructor taking a cubiecube instance as its argument and some functions which will set the move tables and prune tables or would retrieve it from a file if they are stored locally.

IOhandler.h

```
void dumpToFile(string filename, vector<int> data);
void dumpToFile(string filename, vector<vector<int>> data);
void readFromFile(string filename, vector<int> &data, int size);
void readFromFile(string filename, vector<vector<int>> &data,
                        int row, int col);
bool fileExists(string filename);
```

This file contain three functions where the dumpToFile and readFromFile is overloaded so to make it more generic. These function have just the role of dumping and reading data from a binary file stored locally and fileExists is a function that can be used to check if the file exists locally or not.

Search.h

```
class Search{
public:
    vector<int> ax;
    vector<int> po;
    vector<int> flip;
    vector<int> twist;
    vector<int> slice;
    vector<int> parity;
    vector<int> URFtoDLF;
    vector<int> FRtoBR;
    vector<int> URtoUL;
    vector<int> UBtoDF;
    vector<int> URtoDF;
    vector<int> minDistPhase1;
    vector<int> minDistPhase2;
    Search();
    string solutionToString(int length);
    string solution(string facelets, int maxDepth, int timeOut);
    int totalDepth(int depthPhase1, int maxDepth);
};
```

The Search class contains 2 vectors ax and po to denote the axis in {U,R,F,D,L,B} and power to denote if the move is 90^o, 180^o or 270^o. The other 8 vectors just denote the coordinate at the time of that move this is helpful because it may so happen that due to pruning we have to come back to our earlier move. The next two vector denote the minimum distance of the current state to final state. The function solution takes facelet string as its parameter and maxDepth to limit the IDA* search with a timeout. The totalDepth is useful in getting the final depth after applying both phases of the algorithm