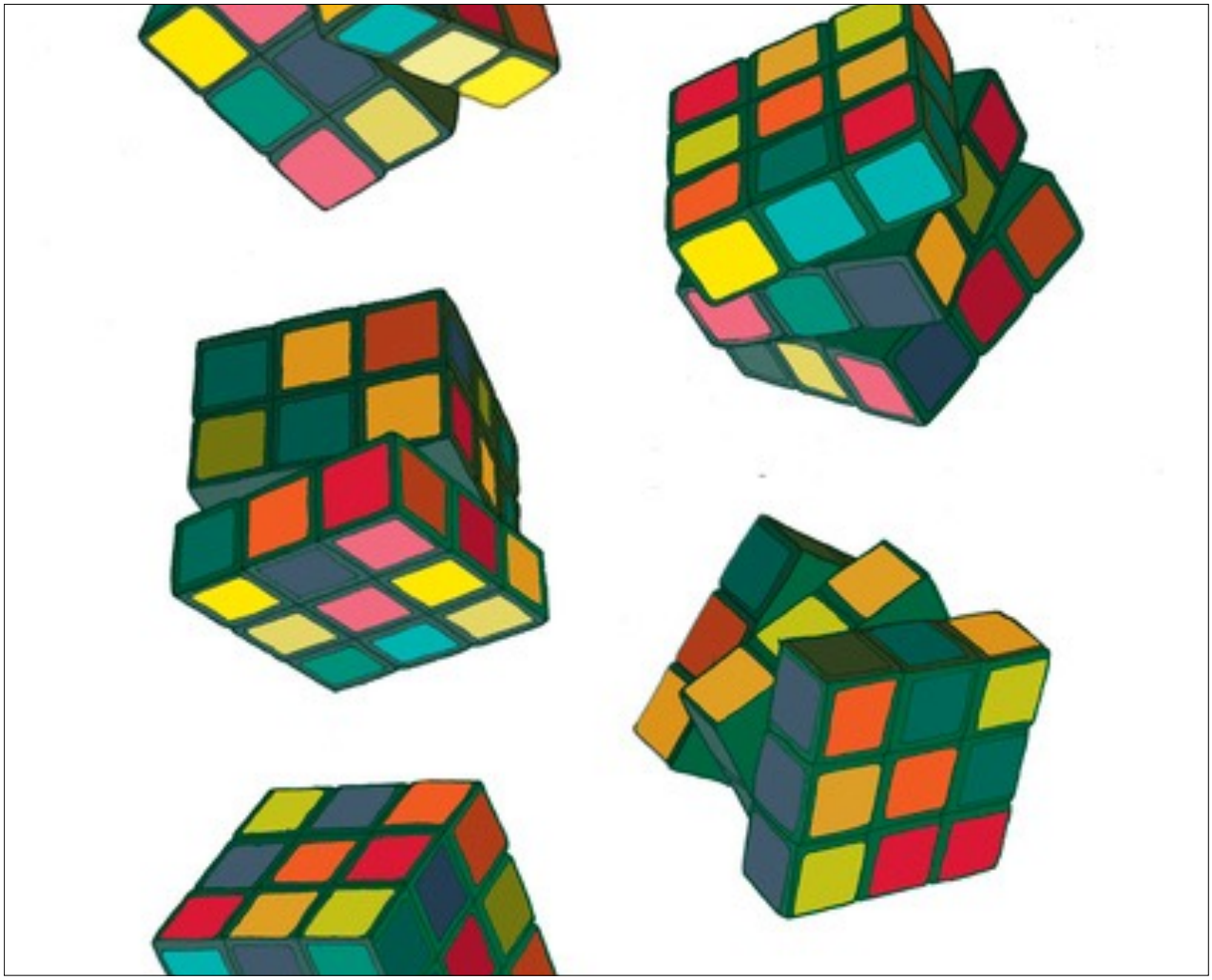


Pocket Cube Solver

Prepared for: CS1005 Mini Project Report

Prepared by: Sayan Das, Saurabh Garg, Shubham Agrawal, Shashwat Hegde

26 October 2015

SUMMARY

Introduction

We all are always excited to solve Rubik's cube in our free time .So to Implement our free time in a useful manner our team decided to work on a project of a Rubik's cube. It is really tough to solve a Rubik's cube specially when it is a 5x5,6x6.....etc. Our project will give a simple defined algorithm by which we can easily solve the Rubik's cube . This project is based on the method of solving Rubik's cube. The project is totally based on object oriented programming(using C++).

Rubiks Cube

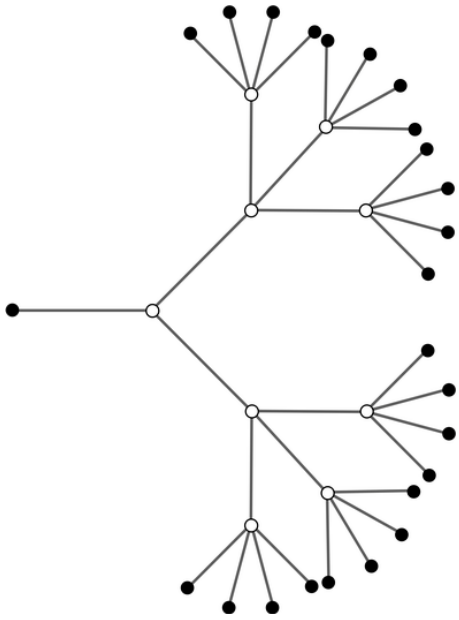
We took 2x2x2 Pocket Cube for this project. There are 8! (40,320) ways to arrange the corner cubes at arbitrary position. Each have their own 3 possible twists. Hence, if we consider number of vertices in the graph we will get 8!*3!(264, 539, 520). We can reduce this by considering cube symmetries by 24 and 3 by considering configurations which are not possible due to it internal structure. So we get (24!*3!)/24!/3 number of vertices to traverse.

Solution

Pocket can be solved by many ways but solving by graph theory is the most efficient way to solve it.

Graph Theory

In mathematics and computer science, graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of vertices or nodes or points and edges or arcs or lines that connect them. A graph may be undirected, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another; see Graph (mathematics) for more detailed definitions and for other variations in the types of graph that are commonly considered. Graphs are one of the prime objects of study in discrete mathematics.



Breadth First Search

Breadth – first searches are performed by exploring all nodes at a given depth before proceeding to the next level. This means that all immediate children of nodes are explored before any of the children's children are considered. It has obvious advantage of always finding a minimal path length solution when one exists. However, a great many nodes may need to be explored before a solution is found, especially if the tree is very full. This approach consumes **O(V+E)** time, where V is the vertices and E is number of edges present in the graph.

```
1  Breadth-First-Search(G, v):
2
3      for each node n in G:
4          n.distance = INFINITY
5          n.parent = NIL
6
7      create empty queue Q
8
9      v.distance = 0
10     Q.enqueue(v)
11
12     while Q is not empty:
13
14         u = Q.dequeue()
15
16         for each node n that is adjacent to u:
17             if n.distance == INFINITY:
18                 n.distance = u.distance + 1
19                 n.parent = u
20                 Q.enqueue(n)
```

Permutation Review

Informally, a permutation describes an ordering of a set's elements. Formally, an N-element per- mutation is a one-to-one mapping of the set 1, 2 . . . N to itself. A permutation f can be represented by the list of N numbers f (1), f (2) . . . f (N). Here is an example of a 5-element permutation:

$$\pi = (34152)$$

Permutations are sometimes described using the explicit notation below. This notation is useful for understanding the inner workings of permutations.

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 2 & 4 \end{pmatrix}$$

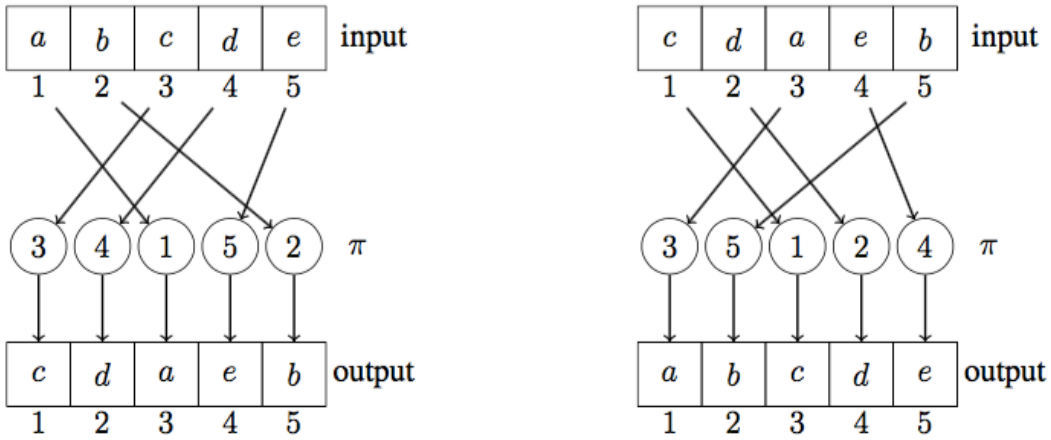


Figure 1: The result of applying π to $[a, b, c, d, e]$. Applying π^{-1} to this result produces the original list.

The inverse of a permutation "undoes" the effects of applying the permutation to a list of elements. For example, by applying π^{-1} (the inverse of π) to [c, d, a, e, b], we should obtain the original list [a, b, c, d, e]. Remember that we obtained [c, d, a, e, b] by applying π to [a, b, c, d, e].

A permutation's inverse can be computed by observing that $\pi^{-1}(\pi(i)) = i$ for $1 \leq i \leq N$. Intuitively, if π moves the third element in the input to the first position in the output, π^{-1} must take the first element in that output (which becomes its input) and move it to the third position in its own output, because π^{-1}'s output must match π's input. Figure 2 illustrates the process of computing π^{-1}

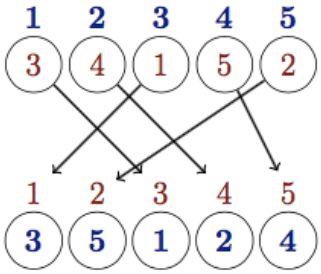


Figure 2: Computing π^{-1}. $\pi(1) = 3$, so $\pi^{-1}(3) = 1$. $\pi(2) = 5$, so $\pi^{-1}(5) = 2$. $\pi(3) = 1$, so $\pi^{-1}(1) = 3$. $\pi(4) = 2$, so $\pi^{-1}(2) = 4$. $\pi(5) = 4$, so $\pi^{-1}(4) = 5$.

Cube State

A plastic 2x2 Rubik's cube is made out of 8 plastic "cubelets". Each plastic cubelet has 3 visible faces that are colored, and 3 faces that are always face the center of the big cube, so we never see them, and we ignore them from now on. Therefore, a plastic 2x2 Rubik's cube has 24 (8x3) colored plastic faces belonging to the 8 cubelets.

The code represents plastic faces using constants named as follows: yob is the yellow face of the cubelet whose visible faces are yellow, orange, and blue. The code also numbers the 24 plastic faces from 0 to 23, and these numbers are the values of the constants named according to the convention above.

One way of representing the Rubik's cube configurations is to reflect the process of building a physical cube by pasting the 24 colored plastic faces on a wireframe of a cube. The left side of Figure 3 shows a wireframe 2x2 Rubik's cube.

The cube's wireframe has 8 cubeletes wireframes, each of which has 3 visible hollow faces where we can paste a plastic face. We refer to the wireframe faces as follows: flu is the front face of the front-left-upper cubelet in the wireframe. Wireframe faces are also associated numbers from 0 to 23.

A cube configuration describes how plastic faces are pasted onto the wireframe cube, so it maps the 24 plastic faces 0-23 to the 24 wireframe faces 0-23. This means that a configuration is a permutation, which can be stored in a 24-element array.

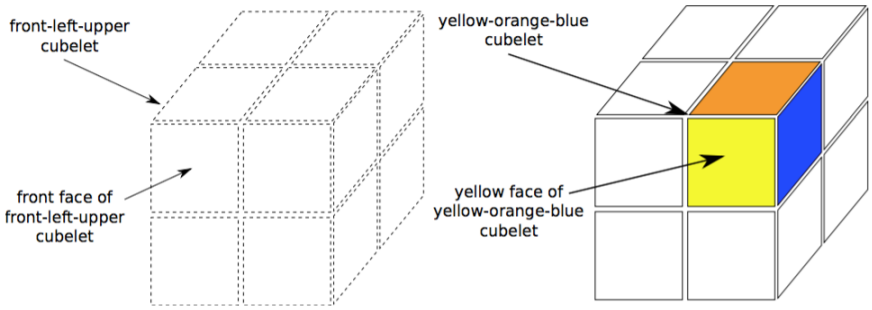


Figure 3: The wireframe Rubik's 2x2 cube is at the left, and the plastic cube is at the right. The plastic cube is made out of plastic faces, and the wireframe cube has positions where the plastic faces can be pasted.

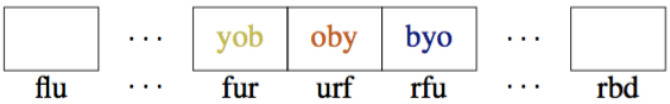


Figure 4: A configuration of the Rubik's cube is represented as a 24-element array, mapping the 24 plastic faces to the 24 wireframe faces. The configuration above has the `yob` plastic cubelet mapped to the `fru` wireframe cubelet, like in Figure 3.

Cube Twist

In order to implement NEIGHBOR(V), we need to analyse the configuration changes caused by cube twists, and code them up inside the NEIGHBOR(V) method.

Twisting a cube changes the cube's configuration, by changing the position of the plastic faces onto the wireframe. A twist moves the cube's wireframe so, for example, rotating the front face clockwise will always move the plastic face that was pasted onto flu (the front face of the front- left-upper cubelet) to rfu (the right face of the front-right-upper cubelet). Figure 5 illustrates the effects of rotating the front face clockwise.

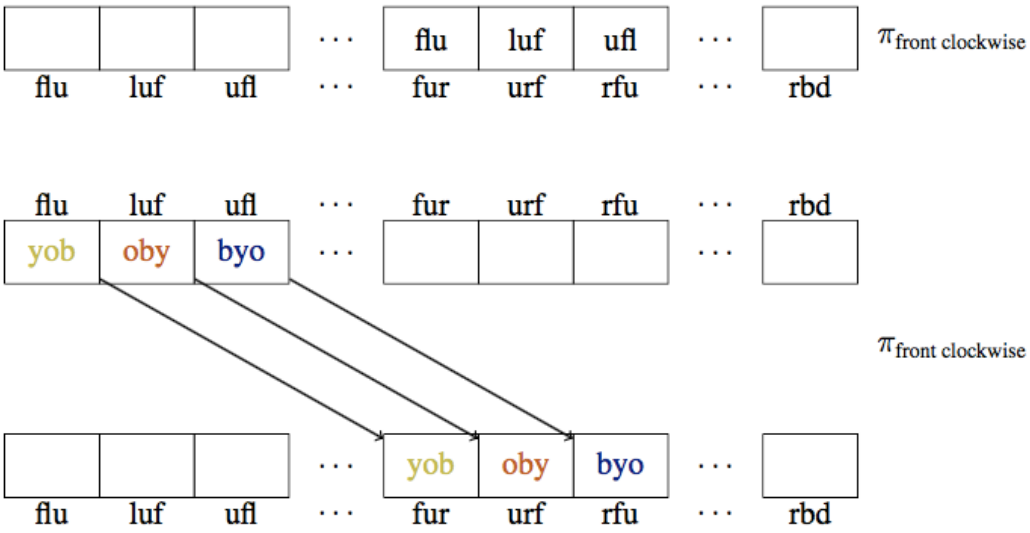


Figure 5: The configuration changes and permutation representing a clockwise twist of the cube's front face.

Due to the property above, we can represent cube twists as permutations. Applying a twist's permutation to the list describing a configuration permutation produces the list describing the new configuration permutation. So we can implement NEIGHBOR(V) by hard-coding the permutations corresponding to cube twists, and applying them to the configuration corresponding to v.

The inverse of a twist's permutation represents the move that would undo the twist.

PROGRAM

Following program is also present on GitHub (**https://github.com/poke19962008/CS1005-Mini-Poject**)

rubiks.h

```
#include <iostream>
#include <map>
#include <string>
#include <array>
using namespace std;

// flu refers to the front face (because f is first) of the cubie that
// has a front face, a left face, and an upper face.
// yob refers to the colors yellow, orange, blue that are on the
// respective faces if the cube is in the solved position.
const int rgw = 0, flu = 0;
const int gwr = 1, luf = 1;
const int wrg = 2, ufl = 2;

const int rwb = 3, fur = 3;
const int wbr = 4, urf = 4;
const int brw = 5, rfu = 5;

const int ryg = 6, fdl = 6;
const int ygr = 7, dlf = 7;
const int gry = 8, lfd = 8;

const int rby = 9, frd = 9;
const int byr = 10, rdf = 10;
const int yrb = 11, dfr = 11;

const int owg = 12, bul = 12;
const int wgo = 13, ulb = 13;
const int gow = 14, lbu = 14;

const int obw = 15, bru = 15;
const int bwo = 16, rub = 16;
const int wob = 17, ubr = 17;

const int ogy = 18, bld = 18;
const int gyo = 19, ldb = 19;
const int yog = 20, dbl = 20;

const int oyb = 21, bdr = 21;
const int ybo = 22, drb = 22;
```

```
const int boy = 23, rbd = 23;

// Front face rotated clockwise.
array<int, 24> F = {fdl, dlf, lfd, flu, luf, ufl, frd, rdf, dfr, fur, urf, rfu,
                    bul, ulb, lbu, bru, rub, ubr, bld, ldb, dbl, bdr, drb, rbd};
// Front face rotated anticlockwise.
array<int, 24> Fi;

// Lower face rotated clockwise.
array<int, 24> L = {ulb, lbu, bul, fur, urf, rfu, ufl, flu, luf, frd, rdf, dfr,
                    dbl, bld, ldb, bru, rub, ubr, dlf, lfd, fdl, bdr, drb, rbd};
// Lower face rotated anticlockwise.
array<int, 24> Li;

// Upper face rotated clockwise.
array<int, 24> U = {rfu, fur, urf, rub, ubr, bru, fdl, dlf, lfd, frd, rdf, dfr,
                    luf, ufl, flu, lbu, bul, ulb, bld, ldb, dbl, bdr, drb, rbd};
// Upper face rotated anticlockwise.
array<int, 24> Ui;

// Identity: equal to (0, 1, 2, ..., 23).
// Final destination position
array<int, 24> I = {flu, luf, ufl, fur, urf, rfu, fdl, dlf, lfd, frd, rdf, dfr,
                    bul, ulb, lbu, bru, rub, ubr, bld, ldb, dbl, bdr, drb, rbd};

// Key-Value -> 'F'|'Fi'|'L'|'Li'|'U'|'Ui' : F|Fi|L|Li|U|Ui
map <string, array<int, 24> > quarter_twist;

// Key-Value -> F|Fi|L|Li|U|Ui : 'F'|'Fi'|'L'|'Li'|'U'|'Ui'
map <array<int, 24>, string > quarter_twist_name;

class rubik {
private:
    // Inverse permuatation for all anticlockwise moves
    array<int, 24> perm_inverse(array<int, 24> X){
        array<int, 24> Xi;
        for(int i=0;i<24;i++) Xi[X[i]] = i;
        return Xi;
    }

```

```
public:
    rubik(){
        // Set all the anticlockwise move
        Fi = perm_inverse(F);
        Li = perm_inverse(L);
        Ui = perm_inverse(U);

        // Map all quarter moves
        quarter_twist["F"] = F;
        quarter_twist["Fi"] = Fi;
        quarter_twist["L"] = L;
        quarter_twist["Li"] = Li;
        quarter_twist["U"] = U;
        quarter_twist["Ui"] = Ui;

        // Inverse mapping of quarter move
        quarter_twist_name[F] = "Front Clockwise";
        quarter_twist_name[Fi] = "Front Anticlockwise";
        quarter_twist_name[L] = "Left Clockwise";
        quarter_twist_name[Li] = "Left Anticlockwise";
        quarter_twist_name[U] = "Upper Clockwise";
        quarter_twist_name[Ui] = "Upper Anticlockwise";
    }

    // Apply permutaion on the current position
    array<int, 24> perm_apply(string move, array<int, 24> postion){
        array<int, 24> perm = quarter_twist[move];
        array<int, 24> newPosition;

        for(int i=0;i<24;i++) newPosition[i] = postion[perm[i]];
        return newPosition;
    }

    // Return move array by move name
    array<int, 24> get_move(string move){
        return quarter_twist[move];
    }

    // Return move name by move array
```

```
    string get_move_name(array<int, 24> perm){
        return quarter_twist_name[perm];
    }

    array<int, 24> get_final_position(){
        return I;
    }

};


```

solver.cpp

```cpp
#include <iostream>
#include <array>
#include <vector>
#include <string>
#include <map>
#include "rubik.h"
using namespace std;

rubik rubiks;

// 1 Way BFS Algorithm
vector<string> shortest_path(array<int, 24> ini, array<int, 24> fin, bool debug_mode){
    int L=1;
    bool found = false;
    vector<string> solution = {};

    if(ini == fin){
        solution.push_back("SOLVED_STATE");
        return solution;
    }

    if (debug_mode)
        cout<<"Debuging Mode\n";

    struct parent_node{
        array<int, 24> pos;
        string move;
    };

    map<array<int, 24>, int> level;
    level[ini] = 0;

    map<array<int, 24>, parent_node > parent;
    parent[ini].pos = {};
    parent[ini].move = "NULL";

    vector< array<int, 24> > frontier = {ini};

    while(frontier.size() && !found){
        vector< array<int, 24> > next = {};
```

```
41
42          for (int i = 0; i < frontier.size() && !found; ++i){
43              vector<string> moves = {"F", "Fi", "L", "Li", "U", "Ui"};
44
45              for (int j = 0; j < moves.size(); ++j){
46                  array<int, 24> next_move = rubiks.perm_apply(moves[j], frontier[i]);
47
48                  if(next_move == fin){
49                      found = true;
50                      parent[next_move].pos = frontier[i];
51                      parent[next_move].move = moves[j];
52                      break;
53                  }
54
55                  if(level.find(next_move) == level.end()){
56                      level[next_move] = L;
57                      parent[next_move].pos = frontier[i];
58                      parent[next_move].move = moves[j];
59                      next.push_back(next_move);
60                  }
61              }
62          }
63          if(debug_mode)
64              cout<<"Level="<<L<<"    Frontier Nodes="<<frontier.size()<<endl;
65          frontier = next;
66          L++;
67      }
68
69      if(found){
70          parent_node tmp = parent[fin];
71
72          while (true){
73              if(tmp.move == "NULL") break;
74              solution.insert(solution.begin(), tmp.move);
75              tmp = parent[tmp.pos];
76          }
77
78      }
79
80      return solution;
```

```
}

int main(){

    // Test Case #1 [TESTED]
    // God's Number: 14
    // Solution :
        // Front Clockwise
        // Front Clockwise
        // Upper Anticlockwise
        // Left Clockwise
        // Upper Anticlockwise
        // Front Clockwise
        // Left Anticlockwise
        // Upper Clockwise
        // Left Anticlockwise
        // Front Anticlockwise
        // Left Anticlockwise
        // Upper Clockwise
        // Left Anticlockwise
        // Front Anticlockwise
    // array<int, 24> ini = {6, 7, 8, 20, 18, 19, 3, 4, 5, 16, 17, 15, 0, 1, 2, 14, 12, 13, 10, 11, 9, 21, 22, 23};

    // Test Case #2 [TESTED]
    // God's Number: 2
    // Solution:
        // Left Anticlockwise
        // Front Anticlockwise

    // Test Case #3 [ ]
    // God's Number:
    // Solution:
    // array<int, 24> ini = {7, 8, 6, 20, 18, 19, 3, 4, 5, 16, 17, 15, 0, 1, 2, 14, 12, 13, 10, 11, 9, 21, 22, 23};

    array<int, 24> ini = rubiks.perm_apply("F", rubiks.get_final_position());
    ini = rubiks.perm_apply("L", ini);

    array<int, 24> fin = rubiks.get_final_position();

    bool debug_mode = false;
```

```
    vector<string> solution = shortest_path(ini, fin, debug_mode);

    if(solution.size() == 0 )
        cout<<"\nSorry you have entered wrong combination...";
    else if(solution[0] == "SOLVED_STATE"){
        cout<<"\nSolved State...";
    }else{
        cout<<"\nSolution: \n";
        for (int i = 0; i < solution.size(); ++i){
            cout<<rubiks.get_move_name(rubiks.get_move(solution[i]))<<endl;
        }
    }
    return 0;
}
```