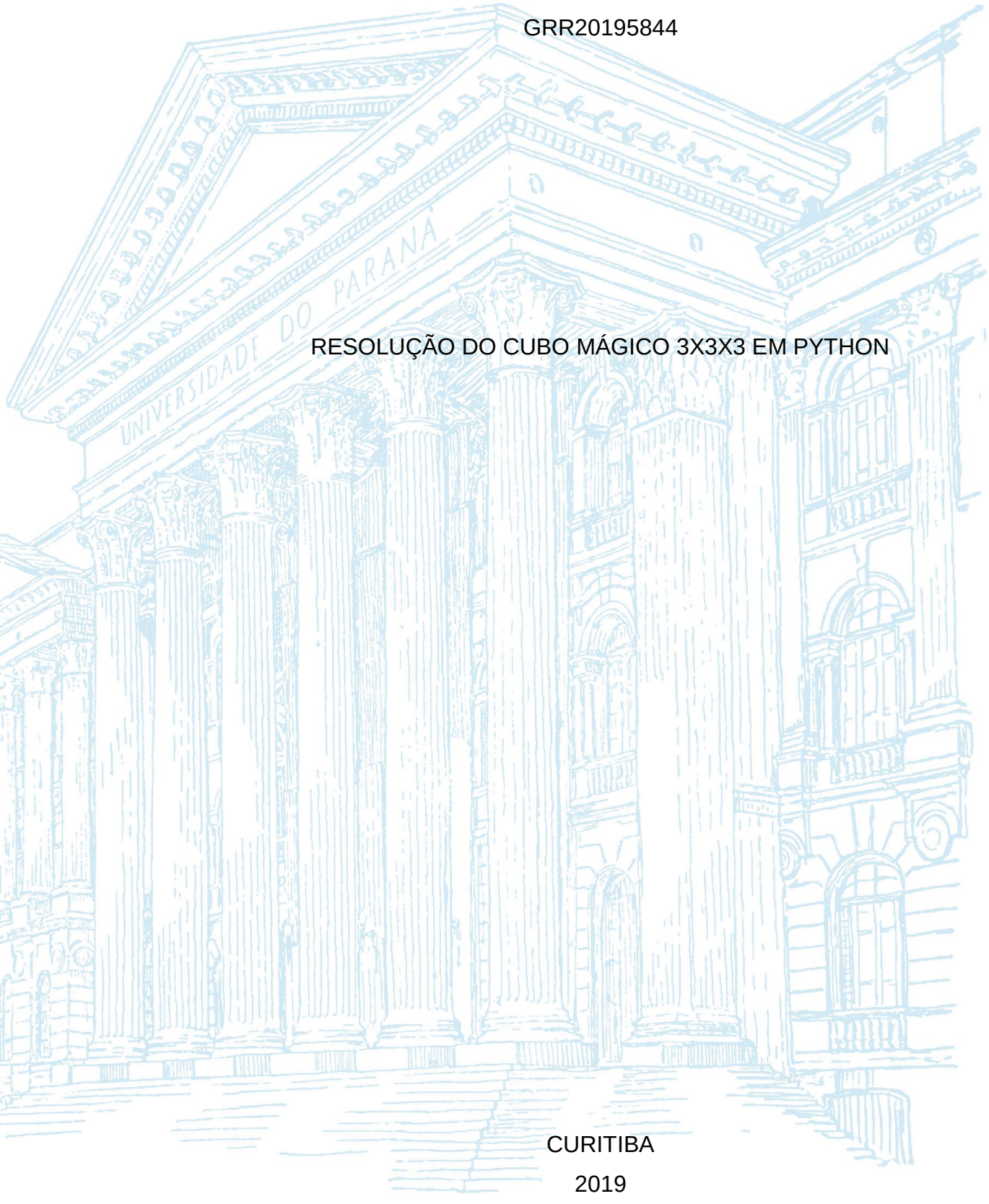
UNIVERSIDADE FEDERAL DO PARANÁ

ISAC MESSIAS MICHELON

GRR20195844



RESOLUÇÃO DO CUBO MÁGICO 3X3X3 EM PYTHON

CURITIBA

2019

ISAC MESSIAS MICHELON

RESOLUÇÃO DO CUBO MÁGICO 3X3X3 EM PYTHON

CIDADE
2019

RESUMO

O cubo mágico 3x3x3 é o brinquedo mais vendido e mais famoso do mundo, porém carrega consigo grande complexidade. Nesse sentido, o propósito do trabalho foi criar um programa em python que resolvesse o cubo. O programa soube aplicar efetivamente um embaralhamento dado e, usando de várias funções que mimicam a resolução humana dada por um algoritmo definido, resolver o cubo a partir desse embaralhamento. O programa recebe a sequência de movimentos que embaralharam o cubo e devolve sua resolução, além de uma contagem de quantos movimentos foram gastos até o estágio resolvido. O programa é consideravelmente eficiente, resolvendo o cubo com uma média de 105 movimentos.

SUMÁRIO

1 INTRODUÇÃO

2. FUNDAMENTAÇÃO TEÓRICA

3. METODOLOGIA

Não foi utilizada nenhuma biblioteca previamente criada para esse trabalho.

3.1 Funções e features criadas:

Nesse trabalho foram criadas as funções:

-mostra
-mostra_cubo
-turn
-alg
-trans_2p3
-find
-cruz
-cantos1
-seg_cam
-cruz_amarela
-oll_de_cruz
-pll
-Termina_cubo
-tratamento
-monta_cubo
-traducao_IpG
-traducao_GpI
-Classe meio
-Classe canto
-Dicionário cubo

Vamos ver o que cada uma faz passo-a-passo:

```
def mostra(v):
```

A função mostra é uma função que mostra na tela uma matriz v NxN.

Ela cria uma string com os elementos da matriz, de forma que os elementos da linha a1j são separados por espaço e as linhas são separadas por uma quebra de linha.

```
def mostra_cubo(cubo):
```

A função mostra_cubo é uma função que mostra na tela as faces do cubo, ou, mais genericamente, que imprime uma lista de K matrizes quadradas, uma embaixo da outra.

A função mostra_cubo é a maneira de mostrar o cubo de modo um pouco mais fácil, menos confuso.

```
cubo = {'Y': [],
        'B': [],
        'O': [],
        'G': [],
        'R': [],
        'W': []}
```

O dicionário cubo aqui é criado, de modo que, invés de se referir as posições de cubo como cubo[0], cubo[1], etc, será utilizado a notação cubo[“Y”], que representa a face amarela. Essa mudança de representação é de grande importância para facilitar o entendimento do código.

```
for k,v in cubo.items():
    for i in range(3):
        v.append([])
        for j in range(3):
            v[i].append(k)
```

Nessa parte do programa o dicionário cubo é preenchido com informações referentes a posição resolvida do cubo, então inicialmente toda posição cubo[“Y”][i][j]; i,j = 1, 2, 3; será igual a “Y”

```
def turn(X, cubo):
```

A função turn leva como parâmetros uma letra e um vetor.

A letra representa qual movimento será realizado, segundo essa equivalência:

```
    NOTAÇÃO P/ PROGRAMA    ###       NOTAÇÃO USUAL
    -    Y                 ###             U
```

```
    -    U                ###            U'
    -    B                ###            F
    -    N                ###            F'
    -    O                ###            L
    -    P                ###            L'
    -    G                ###            B
    -    H                ###            B'
    -    R                ###            R
    -    T                ###            R'
    -    W                ###            D
    -    E                ###            D'
```

As rotações ocorrem em duas partes:

A primeira é rotacionar a face:

```
      cubo['Y'][0][2], cubo['Y'][1][2], cubo['Y'][2][2], cubo['Y'][0][1], cubo
['Y'][2][1], cubo['Y'][0][0], cubo['Y'][1][0], cubo['Y'][2][0] = cubo['Y'][0][
0], cubo['Y'][0][1], cubo['Y'][0][2], cubo['Y'][1][0], cubo['Y'][1][2], cubo['
Y'][2][0], cubo['Y'][2][1], cubo['Y'][2][2]
```

Nessa linha o programa rotaciona a face amarela em -90°.

Ela é a mesma para todos os movimentos no sentido horário, não depende da face.

A lógica dessa linha é que:

A cor da posição cubo[“Y”][0][0] vai acabar na posição cubo[“Y”][0][2], então a posição cubo[“Y”][0][2] recebe a posição cubo[“Y”][0][0].

```
      cubo['Y'][2][0], cubo['Y'][1][0], cubo['Y'][0][0], cubo['Y'][2][1], cubo
['Y'][0][1], cubo['Y'][2][2], cubo['Y'][1][2], cubo['Y'][0][2] = cubo['Y'][0][
0], cubo['Y'][0][1], cubo['Y'][0][2], cubo['Y'][1][0], cubo['Y'][1][2], cubo['
Y'][2][0], cubo['Y'][2][1], cubo['Y'][2][2]
```

Nessa linha o programa rotacional a face amarela em 90°.

Da mesma forma que para rotações no sentido horário, esse movimento não depende da face.

O segundo tipo de movimentação é o das cores adjacentes as que estão girando, que terminam de representar a movimentação da face.

```
for i in range(3):
            cubo['Y'][-i-1][2], cubo['G'][i][0], cubo['W'][-i-
1][2], cubo['B'][-i-1][2] = cubo['B'][-i-1][2], cubo['Y'][-i-
1][2], cubo['G'][i][0], cubo['W'][-i-1][2]
```

As movimentações desse tipo têm essa cara, e trocam elementos de colunas e linhas em volta da face desejada. Nesse exemplo, estão sendo trocadas as colunas 2 da matriz amarela com a 0 da matriz verde com a 2 da matriz azul com a 2 da matriz branca. Elas trocam as entradas ordenadamente.

Essa função tem grande importância para o programa, pois além de movimentar o cubo, é ela que representa a orientação do cubo. Ou seja, como foi definido que a face de cima é a amarela, quando faz um movimento do tipo Y, equivalente ao movimento do tipo U convencional, é a face amarela que deve rotacionar, e as primeiras linhas das matrizes “R”, “B”, “G” e “O” também.

```
def alg(v, cubo):
    #Essa função pega uma string contendo movimentos, transforma ela num vetor
 e aplica as sucessivas rotações descritas nessa string
    v.split()
    for i in range(len(v)):
        turn(v[i], cubo)
    return cubo
```

A função alg leva como parâmetro uma string e um vetor, e serve para facilitar a escrita do código.

Ela serve para realizar vários movimentos seguidos, então se eu quero realizar os movimentos “U” e depois “N”, eu não preciso escrever

turn(“U”, cubo)

turn(“N”, cubo)

Basta escrever

alg(“UN”, cubo) .

```
class canto:
```

A classe canto serve como ferramenta para futuras comparações

```
def __init__(self, x, y, z):
```

```
        self.x = x
        self.y = y
        self.z = z
```

Essa parte do código define que todo canto deve conter 3 informações, sua cor em relação ao eixo x, ao eixo y e ao eixo z.

Por definição, o eixo x é o da face azul, o eixo y é o da face vermelha e o eixo z é o da face amarela.

```
def __eq__(self, other):
        return (self.x == other.x and self.y == other.y and self.z == other.z)
 or (self.x == other.x and self.y == other.z and self.z == other.y) or (self.x
 == other.y and self.y == other.x and self.z == other.z) or (self.x == other.y
 and self.y == other.z and self.z == other.x) or (self.x == other.z and self.y
 == other.y and self.z == other.x) or (self.x == other.z and self.y == other.x
 and self.z == other.y)
```

Essa parte do código define o que é igualdade para elementos da classe meio e será de importância vital para a resolução do cubo.

```
class meio:
```

A classe meio tem papel parecido que a classe canto, mas representa peças diferentes.

```
    def __init__(self, cor1, cor2):
        self.cor1 = cor1
        self.cor2 = cor2
```

Essa parte do código define que cada meio deve ter 2 cores, e a ordem dessas cores vai ser importante para resolver o cubo.

```
def __eq__(self, other):
        return (self.cor1 == other.cor1 and self.cor2 == other.cor2) or (self.
cor1 == other.cor2 and self.cor2 == other.cor1)
```

Essa parte tem o mesmo intuito da igualdade de classes de canto.

```
def trans_2p3(cubo):
```

A função trans2_3 tem como parâmetro uma lista e devolve 20 valores, referentes aos 8 cantos e 12 meios do cubo.

Ela traduz a forma de 6 matrizes para elementos de cantos e meios, o que facilita a resolução do cubo posteriormente.

A ordem dos cantos é definida de modo que o canto 1 seja o canto da parte de cima, de trás e na esquerda. Daí segue um Z até o canto de cima na frente e na direita ser o canto 4.

Virando o cubo em 180° no eixo y e repetindo o processo, se chega na numeração dos cantos.

A ordem dos meios é da seguinte forma:

O meio 1 fica na face amarela e verde, o M2 na face amarela e laranja, o M3 na face amarela e vermelha e o M4 na face amarela e azul.

Os meios M5 ao M8 são definidos de modo que o M5 é o azul e laranja, e cada nodo se dá ao rotacionar o cubo como um todo em -90° no eixo z.

O meios M9 ao M12 seguem a mesma lógica que os meios M1 ao M4, basta rotacionar o cubo 180° no eixo y.

```
C1 = canto(cubo["G"][0][2], cubo["O"][0][0], cubo["Y"][0][0])
```

Exemplo do canto C1 sendo construído

```
M6 = meio(cubo["B"][1][2], cubo["R"][1][0])
```

Exemplo do canto M6 sendo construído

```
def find(X, op, cubo):
```

A função find é simples de construir, mas possivelmente a mais importante de todas. Ela tem como parâmetros um objeto X (que deve ser ou

um meio ou um canto), uma string op, que diz se o objeto é um meio ou um canto, e leva o cubo ao qual ela deve compara.

Desse modo, se você procurar um canto de um cubo no próprio cubo, você não receberá nenhuma informação útil, logo deve-se sempre comparar os cantos ou meios do cubo resolvido e encontrar onde eles estão no cubo embaralhado.

```
if op == "C":
        if X == C1:
            return C1
```

Nessa parte do código podemos ver um exemplo.

Se o operador disser que devemos procurar um canto, então a função vai comparar o canto desejado com os cantos no cubo embaralhado e dar em qual canto do cubo embaralhado esse canto está.

```
def cruz(cubo):
```

A função cruz é a primeira função que realmente resolve o cubo. Ela vai receber o cubo embaralhado, procurar o meio desejado nesse cubo embaralhado e dependendo daonde ele estiver vai retornar como o meio deve ir para a posição desejada

```
 OP = find(Mr9, "M", cubo)
    if OP == M1:
        if OP.cor1 == "W":
            move = "YYBB"
            solve += move
            alg(move, cubo)
        elif OP.cor1 == "B":
            move = "UONP"
            solve += move
            alg(move, cubo)
    elif OP == M2:
        if OP.cor1 == "W":
            move = "UBB"
            solve += move
            alg(move, cubo)
        elif OP.cor1 == "B":
            move = "ONP"
            solve += move
            alg(move, cubo)
```

Nesse exemplo, se o meio M9 estiver na posição do meio M2, e estiver orientado com a cor azul para cima, deve-se aplicar os movimentos O, N e P para fazer ele chegar na posição M9 orientado.

A função cruz coloca no lugar os meios M9 até M12.

Essa informação é guardada na variável solve, que vai caminhar por todas as funções.

```
def cantos1(solve, cubo):
```

A função cantos1 resolve os cantos da face branca, usndo uma abordagem parecida, porém um pouco diferente da função cruz.

A função cantos1, e todas as outras funções que efetivamente resolvem o cubo, a partir de agora recebem uma string, a solve, e uma matriz, como parâmetros.

```
D = "RYT"
    E = "NUB"
    M = "RYYTU" + D
```

Essa parte do código salva movimentos pré determinados que se repetem várias vezes para facilitar a escrita do código.

```
def poe(X, solve, cubo):
        if X.x == "W":
            solve += E
            alg(E, cubo)
            return E
        elif X.y == "W":
            solve += D
            alg(D, cubo)
            return D
        elif X.z == "W":
            solve += M
            alg(M, cubo)
            return M
```

Essa função poe serve para colocar um canto na posição M4 com uma cor branca na posição M6 com a cor branca na face branca. Para essa função, sempre colocaremos o canto na posição m4 e onde ele deve ir na posição M6 e então aplicaremos a função poe, e assim colocando o canto no lugar.

```
elif OP == C2:
        move = "WY"
        alg(move, cubo)
        C1, C2, C3, C4, C5, C6, C7, C8, M1, M2, M3, M4, M5, M6, M7, M8, M9, M1
0, M11, M12 = trans_2p3(cubo)
        op1 = poe(C4, solve, cubo)
        solve += move + op1
        move = "E"
        solve += move
        alg(move, cubo)
```

Aqui está um exemplo da função cantos1.

Se o canto estiver na posição C2, ele deve ir para a posição C4 e seu lugar de destino na posição C6 (O primeiro move faz isso), então é aplicada a função poe no cubo e depois se volta os movimentos que foram feitos para arrumar os cantos de modo que se possa usar a função poe.

```
def seg_cam(solve, cubo):
```

A função seg_cam coloca os meios que não são nem brancos nem amarelos no lugar, usando a variação de um mesmo algoritmo.

Esse algoritmo por definição coloca o meio M2 na posição do meio M6 (é denotado O-br, ou seja, da face laranja para as faces azul e vermelho), mas alterando a orientação dele é arrumada a 2 camada.

```
    if OP == M1:
        #Levando em conta a orientação, a peça é colocada no lugar
        if OP.cor1 == "B":
            move = "BUNONPB"
            #G-ob
            solve += move
            alg(move, cubo)
```

Aqui temos o exemplo que o meio M1 vai para a posição do meio M5

```
def cruz_amarela(solve, cubo):
```

A função cruz_amarela arruma os meios amarelos, de modo que se faça uma cruz na face amarela.

São poucos os casos a serem considerados pela própria natureza do cubo, eles levam em conta quantos meios já estão orientados e aonde eles estão.

```
    if cubo["Y"][0][1] == "Y":
        if cubo["Y"][1][0] == "Y" and cubo["Y"][2][1] != "Y":
            move = "BYRUTN"
            solve += move
            alg(move,cubo)
        elif cubo["Y"][1][2] == "Y" and cubo["Y"][1][0] != "Y":
            move = "OYBUNP"
            solve += move
            alg(move,cubo)
        elif cubo["Y"][2][1] == "Y" and cubo["Y"][1][0] != "Y":
            move = "OBYNUP"
            solve += move
            alg(move,cubo)
```

No código acima vemos quais os movimentos devem ser feitos no caso que o meio M1 já esteja corretamente orientado.

```
def oll_de_cruz(solve, cubo):
```

A função oll_de_cruz orienta toda a face amarela.

Ela tem 7 casos.

```
if (alg("Y",cubo)["Y"][2][0] == "Y") and (cubo["G"][0][2] == "Y") and (cubo["R
"][0][2] == "Y") and (cubo["B"][0][2] == "Y"):
        move = "RYTYRYYT"
        solve += "Y" + move
        alg(move, cubo)
```

Aqui é um exemplo de uma verificação de oll.

O programa vai verificar pontos específicos, críticos do cubo para ver qual algoritmo deve ser utilizado.

Nessa fase, é preciso saber aonde nos cantos as cores amarelas estão, e então a face amarela será orientada.

Pórem existe uma peculiaridade nessa parte do código:

```
    elif (alg("Y",cubo)["Y"][2][0] == "Y") and (cubo["Y"][0][0] == "Y") and (
cubo["B"][0][2] == "Y") and (cubo["G"][0][0] == "Y"):
        move = "TNOBRNPB"
        solve += "Y" + move
        alg(move, cubo)
    elif (alg("Y",cubo)["Y"][2][0] == "Y") and (cubo["Y"][0][0] == "Y") and (c
ubo["B"][0][2] == "Y") and (cubo["G"][0][0] == "Y"):
        move = "TNOBRNPB"
        solve += "YY" + move
        alg(move, cubo)
    elif (alg("Y",cubo)["Y"][2][0] == "Y") and (cubo["Y"][0][0] == "Y") and (c
ubo["B"][0][2] == "Y") and (cubo["G"][0][0] == "Y"):
        move = "TNOBRNPB"
        solve += "U" + move
        alg(move, cubo)
    elif (alg("Y",cubo)["Y"][2][0] == "Y") and (cubo["Y"][0][0] == "Y") and (c
ubo["B"][0][2] == "Y") and (cubo["G"][0][0] == "Y"):
        move = "TNOBRNPB"
        solve += move
        alg(move, cubo)
```

Essa cadeia de elif pode parecer, com razão, exatamente igual, porém o que acontece não é igual.

No começo de cada verificação a função alg é ativada, logo o cubo muda. Mesmo que a verificação seja a mesma, o cubo não é. Desse modo o cubo faz movimentos “automaticamente” para ver se ele se encaixa em alguma das condições para se orientar a face amarela.

Esses movimentos “automáticos” são computados na variável solve, mas não são aplicados novamente na variável "move” pois o cubo já se moveu.

```
def pll(solve, cubo):
```

A função pll permuta as peças e basicamente termina de montar o cubo, mas para isso deve ocorrer várias verificações:

```
if ((alg("Y", cubo)["B"][0][0] == cubo["B"][0][2]) and

        (cubo["O"][0][1] == cubo["B"][0][2]) and
        #arruma a parte laranja
        (cubo["O"][0][0] == cubo["O"][0][2]) and
        (cubo["R"][0][1] == cubo["O"][0][2]) and
        #arruma a parte vermelha
```

```
        (cubo["R"][0][0] == cubo["R"][0][2]) and
        (cubo["B"][0][1] == cubo["R"][0][2]) and
        #arruma a parte verde
        (cubo["G"][0][0] == cubo["G"][0][2]) and
        (cubo["G"][0][1] == cubo["G"][0][2])):
        move = "RURYRYRUTURR"
        solve += "Y" + move
        alg(move, cubo)


    elif ((alg("Y", cubo)["B"][0][0] == cubo["B"][0][2]) and
        (cubo["O"][0][1] == cubo["B"][0][2]) and
        #arruma a parte laranja
        (cubo["O"][0][0] == cubo["O"][0][2]) and
        (cubo["R"][0][1] == cubo["O"][0][2]) and
        #arruma a parte vermelha
        (cubo["R"][0][0] == cubo["R"][0][2]) and
        (cubo["B"][0][1] == cubo["R"][0][2]) and
        #arruma a parte verde
        (cubo["G"][0][0] == cubo["G"][0][2]) and
        (cubo["G"][0][1] == cubo["G"][0][2])):
        move = "RURYRYRUTURR"
        solve += "YY" + move
        alg(move, cubo)

    elif ((alg("Y", cubo)["B"][0][0] == cubo["B"][0][2]) and
        (cubo["O"][0][1] == cubo["B"][0][2]) and
        #arruma a parte laranja
        (cubo["O"][0][0] == cubo["O"][0][2]) and
        (cubo["R"][0][1] == cubo["O"][0][2]) and
        #arruma a parte vermelha
        (cubo["R"][0][0] == cubo["R"][0][2]) and
        (cubo["B"][0][1] == cubo["R"][0][2]) and
        #arruma a parte verde
        (cubo["G"][0][0] == cubo["G"][0][2]) and
        (cubo["G"][0][1] == cubo["G"][0][2])):
        move = "RURYRYRUTURR"
        solve += "U" + move
        alg(move, cubo)

    elif ((alg("Y", cubo)["B"][0][0] == cubo["B"][0][2]) and
        (cubo["O"][0][1] == cubo["B"][0][2]) and
        #arruma a parte laranja
        (cubo["O"][0][0] == cubo["O"][0][2]) and
        (cubo["R"][0][1] == cubo["O"][0][2]) and
        #arruma a parte vermelha
        (cubo["R"][0][0] == cubo["R"][0][2]) and
        (cubo["B"][0][1] == cubo["R"][0][2]) and
        #arruma a parte verde
```

```
        (cubo["G"][0][0] == cubo["G"][0][2]) and
        (cubo["G"][0][1] == cubo["G"][0][2])):
        move = "RURYRYRUTURR"
        solve += move
        alg(move, cubo)
```

Aqui é um exemplo de verificação da função pll.

Ela tem a mesma lógica da função oll_de_cruz, fazendo quatro verificações idênticas em instancias diferentes do cubo.

Aqui é comparado posições do cubo, e não cores, pois dependendo da orientação as cores vão mudar, mas esse padrão de igualdade entre as posições não.

Existem 21 casos de pll, o que torna essa função extensa. Sua necessidade de comparar 12 posições 4 vezes para cada caso faz com que ela tenha aproximadamente 1100 linhas de código.

```
def Termina_cubo(solve, cubo):
```

Após todos os passos, a função pll pode deixar o cubo quase resolvido, mas faltando 1 ou 2 movimentos para que ele chege efetivamente no estágio final de resolvido. É esse o papel da função Termina_cubo, ela termina de montar o cubo com 1 ou 2 movimentos na camada amarela.

```
    if cubo["B"][0][0] == "O":
        move = "Y"
        solve += move
        alg(move, cubo)
    elif cubo["B"][0][0] == "G":
        move = "YY"
        solve += move
        alg(move, cubo)
    elif cubo["B"][0][0] == "R":
        move = "U"
        solve += move
        alg(move, cubo)
    return solve
```

Seu funcionamento é simples, ela verifica qual cor está na face azul e indica a partir disso se é necessário movimento adicional, e aplica esse movimento se for necessário.

```
def tratamento(X):
```

Essa função otimiza movimentos desnecessários de dois tipos:

-Movimentos complementares, como U e Y;

-Três movimentos equivalentes a um, como YYY e U.

Como uma otimização pode gerar uma possibilidade de otimização do segundo tipo, e vice-versa, existem 2 variáveis que controlam até quando deve-se verificar se há possibilidade de otimização

Essas variáveis são Z e Y. Elas são construídas como o tamanho do vetor a ser otimizado, mas isso antes de qualquer otimização.

Então digamos que X tenha tamanho 20 e depois de uma otimização ele teve tamanho 19. Pode haver alguma outra otimização a ser feita.

Agora se X tenha tamanho 20 no começo e no final da otimização, então não há mais o que fazer e logo X já está totalmente efetivo.

```
while i < len(X)-1:
            if ((X[i] == "Y" and X[i+1] == "U") or

                (X[i] == "U" and X[i+1] == "Y") or
                (X[i] == "B" and X[i+1] == "N") or
                (X[i] == "N" and X[i+1] == "B") or
                (X[i] == "O" and X[i+1] == "P") or
                (X[i] == "P" and X[i+1] == "O") or
                (X[i] == "G" and X[i+1] == "H") or
                (X[i] == "H" and X[i+1] == "G") or
                (X[i] == "R" and X[i+1] == "T") or
                (X[i] == "T" and X[i+1] == "R") or
                (X[i] == "W" and X[i+1] == "E") or
                (X[i] == "E" and X[i+1] == "W")):
                del X[i : i+1]
                #Se uma otimização foi feita, nenhuma mais será nesse ciclo, p
ara que não haja problemas com índices fora de range
                i += 100
            else:
               i += 1
```

Essa parte do trabalho faz a otimização do tipo 1

```
 Y = len(X)
            i = 0
            while i < len(X)-2:
                if X[i] == "Y" and X[i+1] == "Y" and X[i+2] == "Y":
                    X[i] = "U"
                    del X[i+1 : i+2]
                    i += 100
                else:
                    i += 1
```

Esse é um exemplo de otimização do tipo 2.

```
def monta_cubo(cubo):
    #A função monta cubo reúne cada função e aplica todas elas em ordem, retor
nando o cubo resolvido e a solve que o resolveu
    solve = cruz(cubo)
    solve = cantos1(solve, cubo)
    solve = seg_cam(solve, cubo)
    solve = cruz_amarela(solve, cubo)
    solve = oll_de_cruz(solve, cubo)
    solve = pll(solve, cubo)
    solve = Termina_cubo(solve, cubo)
    solve = tratamento(solve)
    return solve
```

Após tudo isso, a função monta cubo recebe um cubo desmontado e monta ele, juntando todas as funções que fazem isso.

```
def traducao_IpG(X):
```

Além de montar o cubo em sí, existem 2 funções que trabalham com os movimentos de embaralhar, traduzindo eles entre a linguagem global e a do programa.

A primeira é tradução_IpG que recebe uma lista como parâmetro e retorna uma string quase nos padrões globais (ela tem dificuldade de escrever movimentos duplos no padrão usual, ela os representa, mas não no padrão).

```
def traducao_GpI(X):
```

A segunda função recebe uma string como parâmetro e devolve uma string também, pronta para ser aplicada na função alg.

4. RESULTADOS OBTIDOS COM A IMPLEMENTAÇÃO

O programa consegue, na maioria das vezes, resolver o cubo com perfeição. Porém, como cada movimento que o cubo faz baseado em onde a peça está foi escrito manualmente, é possível que eu não tenha percebido possíveis erros que levem o programa a não conseguir resolver o cubo totalmente. Dependendo em qual função esse erro ocorreu, o programa nem chegará perto do estado resolvido do cubo.

Porém, foram realizados vários e vários testes, todos satisfatórios, usando um site online que gera embaralhamentos aleatórios.

Além disso, a quantidade de movimentos mostrou-se eficiente, pois todas as soluções ficaram próximas de 105 movimentos e nenhuma passou de 130 (a modo de comparação profissionais fazem perto de 60 movimentos numa resolução).

Alguns embaralhamentos e respectivas quantidades de movimentos:

- D B' U' B U' D L' B L D2 R D' R' F R' F L' F B' L2 R' U B2 R' U - 93
- U2 L U2 D F2 B L2 D' B' D' B U B F2 D B' U2 F2 D F' B R D' R2 U2 – 100
- B D U2 F R' L2 D U R F' B R2 L2 U L' F' D' R' F L D2 F' R2 D' R – 107
- F D' R' D' F' L' B' D2 U2 L' B F' L' R2 B' L' F R2 L2 F D' U2 R L D – 110
- R U2 D' L D2 B D' B L U D2 R' L' F U D' B U' L D B' L D2 L B' – 99
- F' U F D L2 D' F' R2 L2 F' U R2 F2 R2 L2 F D2 B2 F2 U2 D' F2 R' D F' – 102
- U' R U' F D' F' R2 F' D2 B2 R L D2 R2 B D' L2 R' B' L2 D F R' L' B – 103

5. DIFICULDADES ENCONTRADAS NO TRABALHO

As principais dificuldades do trabalho foram representar o cubo matricialmente, pensar um jeito que não desse muito trabalho para realizar as funções (foi testado umas 3 maneiras, essa foi a que deu certo) e além disso escrever à mão cada caminho que cada peça devia seguir.

Além disso as funções traducao e a função tratamento foram mais complexas de se fazer, principalmente a função tratamento.