

**Fundação Getúlio Vargas  
Escola de Matemática Aplicada**

**Matheus de Moncada Assis**

**Cubo de Rubik: interface gráfica e análise de  
métodos para solução**

Rio de Janeiro  
2020

**Matheus de Moncada Assis**

**Cubo de Rubik: interface gráfica e análise de  
métodos para solução**

Dissertação submetida à Escola de Matemática Aplicada como requisito parcial para a obtenção do grau de Bacharel em Matemática Aplicada.

Orientadora: Asla Medeiros e Sá

Rio de Janeiro  
2020



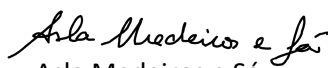
**MATHEUS DE MONCADA ASSIS**

**“CUBO DE RUBIK: INTERFACE GRÁFICA E ANÁLISE DE MÉTODOS PARA SOLUÇÃO”**

Trabalho de Conclusão de Curso - TCC apresentado ao Curso de Graduação em Matemática Aplicada da Escola de Matemática Aplicada para obtenção do grau de Bacharel em Matemática Aplicada.

Data da Defesa: 08/12/2020

**ASSINATURA DOS MEMBROS DA BANCA EXAMINADORA**

  
Asla Medeiros e Sá  
Orientadora

  
Paulo Cezar Pinto Carvalho  
Membro

DocuSigned by:  
  
6812AE5B8AEA41A...  
Alexandre Madureira  
Membro

Nos termos da Lei nº 13.979 de 06/02/20 - DOU nº 27 de 07/02/20 e Portaria MEC nº 544 de 16/06/20 - DOU nº 114 de 17/06/20 que dispõem sobre a suspensão temporária das atividades acadêmicas presenciais e a utilização de recursos tecnológicos face ao COVID-19, as apresentações dos Trabalhos de Conclusão de Curso, de forma excepcional, serão realizadas de forma remota e síncrona, incluindo-se nessa modalidade membros da banca e discente.

## Agradecimentos

Primeiramente, agradeço à toda minha família por sempre me apoiar, em especial à minha mãe, Patricia, por sempre me incentivar e tornar todas as oportunidades que tive na vida possíveis, e às minhas irmãs, Tássia e Jéssica, por sempre me ajudarem durante cada passo dado na vida.

Agradeço também à professora Asla Sá pela ótima orientação científica durante esse projeto e por todo apoio ao longo dos últimos anos.

Por fim, agradeço à Fundação Getúlio Vargas, em especial à Escola de Matemática Aplicada, pela oportunidade de estudar e viver diversas oportunidades durante esses últimos anos.

## Epígrafe

*"We do not stop playing games because we grow old, we grow old because we stop playing."*

KAMIYA, Yuu; **No Game No Life**

## Resumo

O Cubo de Rubik é um dos *puzzles* mais conhecidos do mundo, sendo inclusive um dos mais vendidos na história da humanidade. A dimensão do número de combinações possíveis que ele propõe é um desafio que foi estudado por diversos nomes e, até os tempos atuais, encanta diversos pesquisadores. O trabalho desenvolvido tem o intuito de explorar a sua modelagem e visualização em ambientes computacionais, juntamente com a busca de uma solução através de algoritmos para busca de caminhos. Os resultados são observados e analisados, além de terem seu esforço computacional verificado.

Palavras-chave: Cubo de Rubik, Rubik, interface, algoritmos para busca de caminhos

## **Abstract**

The Rubik's Cube is one of the world's most known puzzles, being one of the best-sellers ever on humanity history. The dimension of the number of possible combinations is a challenge that was studied by several names e, even at current times, delight many researchers. The aim of this work is to explore the Cube's modelling and visualization on computational environments, along with the search for a solution using pathfinding algorithms. All results achieved are observed and analyzed, and their computational efforts verified.

Keywords: Rubik's Cube, Rubik, interface, pathfinding algorithms

# Sumário

<b>1</b>	<b>Introdução</b>	<b>8</b>
1.1	O Desafio . . . . .	8
1.2	A Abordagem . . . . .	9
1.3	O Objetivo . . . . .	10
<b>2</b>	<b>Modelagem</b>	<b>11</b>
2.1	O Código . . . . .	12
2.2	Funções Auxiliares . . . . .	14
<b>3</b>	<b>Visualização</b>	<b>16</b>
3.1	O 3D . . . . .	16
3.2	O 2D abrangente . . . . .	18
3.3	Interface . . . . .	19
3.3.1	Simulação . . . . .	20
<b>4</b>	<b>Soluções</b>	<b>22</b>
4.1	Algoritmo A* . . . . .	22
4.1.1	O Algoritmo . . . . .	23
4.1.2	Resultados do Método . . . . .	24
4.2	Algoritmo IDA* . . . . .	25
4.2.1	Matemática da Função Heurística . . . . .	25

4.2.2	Manhattan Distance 3D para o Cubo de Rubik . . . .	27
4.2.3	O Algoritmo . . . . .	28
4.2.4	Resultados do Método . . . . .	32
<b>5</b>	<b>Conclusão</b>	<b>34</b>
<b>6</b>	<b>Referências</b>	<b>35</b>

# 1 Introdução

Com aproximadamente 43 quintilhões de combinações possíveis, o popular Cubo de Rubik, também conhecido como Cubo Mágico, é um dos *puzzles* mais vendidos na história da humanidade. Criado pelo professor e arquiteto Ernő Rubik em 1974, o Cubo partiu de uma ideia para explicar as relações de espaços para seus alunos, e tomando forma durante as décadas, se tornou um meio de estudo primordial para a matemática e para a computação.

Neste trabalho, temos como objetivo observar essa relação de espaço computacionalmente, abordando questões de visualização e de resolução do Cubo de Rubik. Explorando os limites computacionais, juntamente com a capacidade de modelar e visualizar o Cubo em um computador, buscamos entender e analisar os resultados obtidos.

## 1.1 O Desafio

O Cubo Mágico é formado por vinte e seis peças, sendo seis peças de centro, doze peças de meio, e oito peças de canto. As peças de meio tem uma única cor, as de meio, duas cores, e as de canto, três cores.

Figura 1: Representação de cada peça.



Fonte: Compilação do autor.<sup>1</sup>

---

<sup>1</sup>Montagem a partir de imagens coletadas no site cubovelocidade.com.br.

Dessa forma, o conjunto das peças compões o objeto tridimensional com seis faces coloridas. Ademais, tal formação possibilita a movimentação de todos os lados do cubo, e cada movimento possível rotaciona oito peças ao mesmo tempo.

Partindo de um Cubo resolvido, o mesmo é embaralhado seguindo uma sequência aleatória de  $N$  movimentos e que não é utilizada pelo jogador para solucionar o Cubo seguindo os movimentos inversos. Ao receber o Cubo desorganizado, o objetivo do jogador é retornar o Cubo de volta ao estado original, realizando uma sequência de  $M$  movimentos.

## 1.2 A Abordagem

Ao se deparar com o problema em sua frente, o jogador pode encará-lo de diversas maneiras. A solução mais comum é resolvê-lo pelo método *Layer by Layer*, idealizado por David Singmaster e adaptado por James G. Nourse em seu livro *The Simple Solution to Rubik's Cube* [1]. Com o passar do tempo diversos entusiastas do problema foram capazes de descobrir como solucioná-lo e, aliado ao advento da internet, repassar essa informação se tornou mais simples, fortificando e espalhando a resposta para o mundo.

Outros métodos surgiram ao longo dos anos, mas principalmente com os avanços tecnológicos, o Cubo de Rubik passou a ser decifrado computacionalmente. Sempre explorando os limites, diversos métodos foram surgindo e buscando a resposta da melhor solução possível para ele, no entanto, apenas nos anos mais recentes ela foi alcançada. De tal modo, foi possível encontrar o chamado Número Divino, nomenclatura dada pelos autores ao número mínimo de movimentos para solucionar qualquer estado dele, sendo 20 esse número.<sup>2</sup> Em 2010, Tomas Rokicki, Herbert Kociemba, Morley Davidson, e John Dethridge [2] foram os responsáveis por tal feito.

---

<sup>2</sup>THOMAS ROKICKI. God's Number is 20, c2010. Página única. Disponível em: <<https://www.cube20.org/>>. Acesso em 28 de set. de 2020.

## 1.3 O Objetivo

O objetivo do presente trabalho é de modelar um Cubo de Rubik computacionalmente, explorando soluções automáticas e semiautomáticas para o problema. Para tanto, vamos implementar uma interface gráfica que representa o Cubo bidimensionalmente em ambiente web, bem como implementaremos algoritmos automáticos de solução do Cubo de Rubik em Python.

No segundo capítulo, estaremos estudando a modelagem do Cubo em ambiente computacional, focando em entender suas especificidades e a maneira com que isso afetou os capítulos seguintes. Ademais, o código criado em Python será discutido e entendido, de modo a exemplificar para o leitor a solução escolhida.

No terceiro capítulo, estaremos analisando e entendendo as diversas possíveis visualizações do Cubo de Rubik, além de encaminharmos a escolha da solução desse trabalho. Além disso, também estaremos observando a interface criada para o projeto, sendo ela responsável por unir todas as etapas discutidas nesse trabalho. Tal interface foi criada também com a linguagem Python, mas utilizando principalmente a biblioteca Flask como API para a página web em HTML e CSS.

No quarto capítulo, acerca das soluções, estaremos observando os algoritmos testados no projeto, juntamente com suas falhas e sucessos para a resolução do problema. Também discutiremos as funções heurísticas escolhidas, buscando entender suas necessidades e a maneira com que afetam os algoritmos principais. Finalmente, estaremos verificando o esforço computacional de cada um e as soluções encontradas.

O código desenvolvido para este trabalho em cada uma das etapas está disponível em: <https://github.com/MatheusMAssis/Rubiks-Cube-TCC>

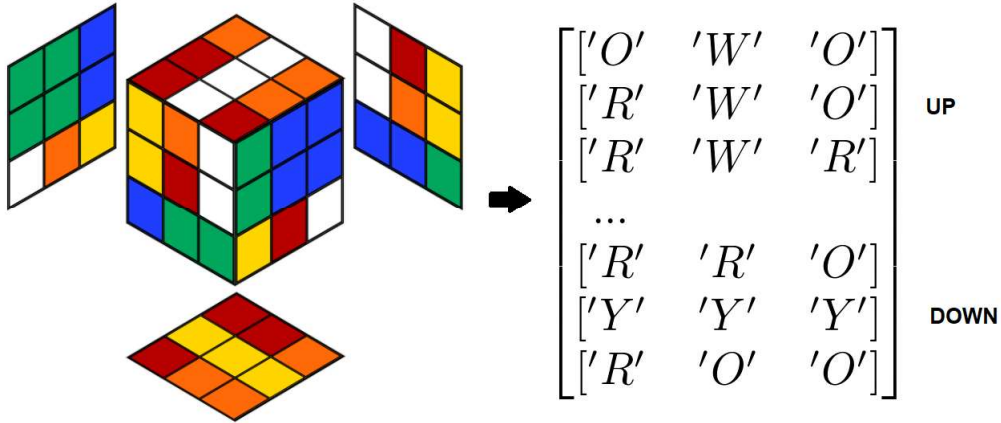
## 2 Modelagem

Por se tratar de um objeto tridimensional, diversas abordagens podem ser tomadas para sua modelagem. Visando facilitar a implementação dos movimentos e tornar a lógica por trás desse Cubo virtual mais acessível, foi escolhida a sua implementação como uma matriz contendo dezoito *arrays* com três elementos. Tal escolha tem como inspiração o projeto de Adrian Liaw, criador da biblioteca PyCuber em Python. [3].

$$\begin{bmatrix} [a_1 & a_2 & a_3] \\ [a_4 & a_5 & a_6] \\ \dots \\ [a_{52} & a_{53} & a_{54}] \end{bmatrix}$$

Diante de tal escolha, é importante levar em conta as escolhas que foram feitas para estabelecer o ambiente do problema. Cada representação nesse formato é única e os centros do Cubo são consequentemente fixos. Podemos observar no exemplo a seguir:

Figura 2: Estado do Cubo modelado.



Fonte: Representação da Modelagem de um estado do Cubo.<sup>3</sup>

Cada elemento  $a_i$  pode conter uma das seis cores possíveis. Dessa forma, fica claro qual cor ocupa tal espaço em determinado momento.

Além de facilitar a implementação, tal modelagem simplifica a principal questão de busca por uma função heurística que veremos mais a frente na seção de Solução. Apesar disso, tal escolha torna possível a manipulação dos dados contidos em cada face do Cubo de maneira simples. Dessa forma, continuaremos analisando a base por trás do código utilizado para criá-lo.

## 2.1 O Código

Para começar qualquer processo nessa etapa de modelagem, foi necessário importar algumas bibliotecas importantes para que o projeto fosse alcançado. São elas: *NumPy* e *Random*. Ambas se destacam por serem naturais da própria linguagem *Python*, além de fornecerem ferramentas de manipulação de dados eficientes e otimizadas.

O *NumPy*, biblioteca de computação científica, é de suma relevância ao tratar de todas as questões matriciais e com escalas numéricas que acabam envolvidas. Já o *Random*, mesmo aparecendo menos ao longo das linhas de código, é essencial para facilitar a geração de aleatoriedades em determinados momentos.

Adentrando nos primeiros passos necessários para definir qualquer estado do Cubo, foi necessário criar uma classe capaz de conter todas as informações necessárias para o Cubo de Rubik. Além disso, tal estrutura deveria ser capaz de lidar com o problema futuro de solução do Cubo. Dessa forma, a primeira classe foi escolhida:

Listing 1: Estrutura da Classe do Cubo

---

```
class State:
    def __init__(self):
        self.cube = None
        self.g    = 0
        self.h    = 0
        self.parent = None
        self.move = None
```

---

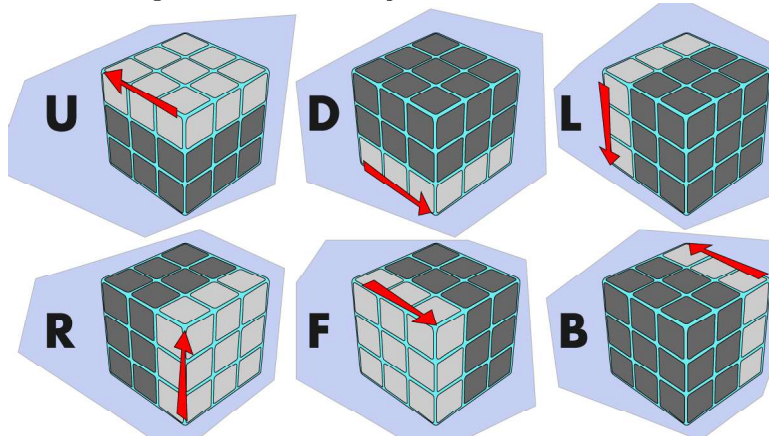
Com ela, visando os algoritmos avançados para encontrar a solução para o Cubo, somos capazes de analisar todas as questões necessárias para avançar

no método, tais como: o estado do Cubo, o seu valor atual de prioridade, o seu estado anterior e o movimento que o fez alcançar tal posição.

Para que fosse possível transitar entre os estados, foi necessária a criação de algumas função de movimento do Cubo. Sabemos, porém, que tais movimentos estão constantemente alterando diversas partes do Cubo simultaneamente e, pensando nisso, tomamos a modelagem anterior como uma ferramenta crucial para tal momento. Acerca de tais movimentos, exploraremos suas especificidades a seguir.

De um modo geral, no Cubo de Rubik 3x3, existem doze movimentos possíveis que não alteram os centros de lugar. Esses movimentos são representados por suas iniciais em inglês: [U]p, [D]own, [R]ight, [L]eft, [F]ront, [B]ack. É importante ressaltar que todos esses movimentos são realizados no sentido horário, enquanto os de sentido anti-horário são representados pelas mesmas iniciais, mas adicionadas de um i: [Ui], [Di], [Ri], [Li], [Fi], [Bi].

Figura 3: Visualização dos movimentos.



Fonte: Representação de cada movimento do Cubo de Rubik.<sup>4</sup>

Para simplificar a visualização do código, disponibilizamos apenas um movimento horário e um anti-horário, mas é de suma importância constatar que os outros movimentos seguem uma lógica similar ao que será apresentado.

<sup>4</sup>Disponível em <<https://hobbylark.com/puzzles/Rubik-Cube-Algorithms/>>. Acesso em 8 de out. de 2020.

---

Listing 2: Estrutura dos Movimentos do Cubo

---

```
def F(arr):
    arr[6:9, 0:3] = np.fliplr(arr[6:9, 0:3].transpose())
    t1 = np.array(arr[2, 0:3])
    t2 = np.array(arr[9:12, 0])
    t3 = np.array(arr[15, 0:3])
    t4 = np.array(arr[3:6, 2])
    arr[2, 0:3] = np.fliplr([t4])[0]
    arr[9:12, 0] = t1
    arr[15, 0:3] = np.fliplr([t2])[0]
    arr[3:6, 2] = t3

def Fi(arr):
    F(arr)
    F(arr)
    F(arr)
```

---

Uma curiosidade acerca desse trecho é a realização do movimento no sentido anti-horário. Por se tratar de um Cubo com movimentos estabelecidos e cíclicos, um movimento no sentido horário pode ser representado por três vezes o movimento no sentido anti-horário e vice-versa. De tal forma, é evidente como os estados do Cubo estão conectados e podem ser descritos de diversas maneiras, mesmo levando mais passos para serem alcançados.

## 2.2 Funções Auxiliares

Entrando no mérito de funções auxiliares ao processo do Cubo, é importante destacar a função de Embaralhamento do mesmo. Para tal, utilizei a biblioteca *Random* do próprio *Python*, a fim de gerar escolhas aleatórias para a sequência do Cubo. Ao escolher um número entre 1 e 12, o movimento associado a tal número é realizado.

Tal função recebe um parâmetro de entrada e ele é utilizado para determinar o número de movimentos que serão escolhidos aleatoriamente. Com isso, temos uma função que embaralha automaticamente o nosso Cubo em ambiente computacional.

### Listing 3: Estrutura do Embaralhamento

---

```
def scramble(arr, n):  
    list_of_movements = []  
    scramble_list = [rd.randint(1, 12) for i in range(n)]  
    for movement in scramble_list:  
        move = make_move(arr, movement)  
        list_of_movements.append(move)  
    return list_of_movements
```

---

## 3 Visualização

Criado com o intuito de explorar o entendimento de espaços, o Cubo de Rubik é um objeto tridimensional e que requer um trabalho de visão espacial concreto para seu entendimento por completo. Qualquer simples movimento afeta diretamente diversas partes do corpo completo simultaneamente, fazendo com que o ponto de referência escolhida seja perdido.

Buscando compreender melhor a atuação de cada movimento em cada parte do Cubo sem perda de informação, foi criada uma visualização que contempla todas as faces em uma tela. De tal modo, é possível acompanhar cada um desses movimentos afetando por completo todas as partes.

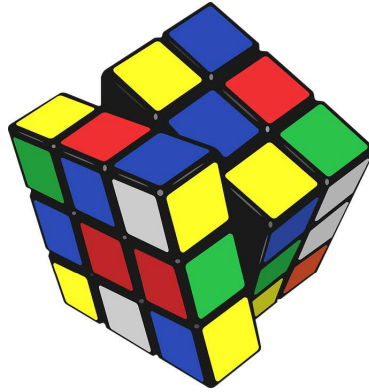
### 3.1 O 3D

A visão que mais estamos acostumados com o Cubo é a tridimensional. Como temos acesso ao objeto no mundo real, movimentá-lo e observar suas transformações a partir de um movimento é mais simples já estando adaptado ao ambiente inserido. No entanto, por estarmos lidando com um computador e com uma tela bidimensional, nem sempre essa é a melhor opção.

Vale ressaltar também a dificuldade para movimentar rapidamente o Cubo nesse ambiente 3D dentro do computador. Mesmo com a utilização do mouse, rotacionar um objeto com ele, e não com as próprias mãos é diferente e pode causar estranhezas. Por tal, no longo prazo, até mesmo para simples objetivos, tal visualização tem diversos pontos críticos com a observação e com as necessidades do usuário ao utilizá-lo.

Além disso, é importante destacar a falta de controle sobre o que você está vendo. Tal situação simples quando pensamos no Cubo em nossas mãos se torna complexa quando o controlamos somente com um mouse. Isso se deve ao fato de perdermos uma referência da posição em que estávamos por um simples deslize, levando o processo de solucioná-lo a um cenário caótico.

Figura 4: Cubo de Rubik tridimensional.



Fonte: Modelo de um Cubo de Rubik.<sup>5</sup>

---

<sup>5</sup>Disponível em <<https://publicdomainq.net/rubiks-cube-0007064/>>. Acesso em 28 de set. de 2020.

### 3.2 O 2D abrangente

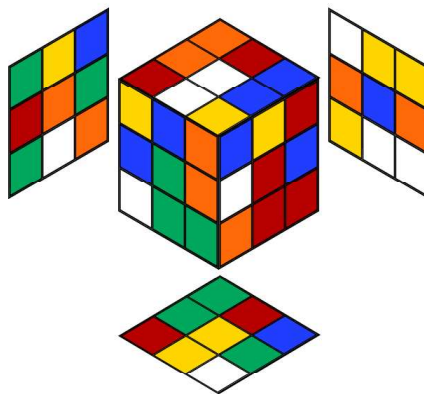
Partindo da falta de controle que temos sobre o espaço tridimensional, é importante ressaltar o quão valiosa é uma ferramenta que consegue mostrar o Cubo e todos seus espaços em uma única tela. Ademais, tal visualização não pode comprometer a espacialidade do Cubo, sendo responsável por manter a perspectiva das peças de acordo com a sua posição.

Uma alternativa que é comumente utilizada é a planificação do objeto, mas tal abordagem torna mais complicado o entendimento sobre os espaços conectados quando determinados acontecem. Exemplificando essa situação, um movimento que envolva as faces frontal e traseira poderia dificultar o entendimento do usuário em relação ao todo.

Por tal motivo, neste trabalho foi implementado a visualização de uma representação do Cubo de Rubik mais abrangente. Dessa forma, ao trabalharmos a solução do mesmo, fica evidente a maneira com que cada peça e cada movimento afeta o estado atual do objeto. Além disso, manter a perspectiva fixa do estado do Cubo facilita a manutenção de um ponto de referência para ele, melhorando o entendimento do comportamento.

Tal escolha de visualização é de autoria própria, mas tem como base uma das visualizações presentes no site <https://rubiks-cube-solver.com/> [4].

Figura 5: Cubo de Rubik 2D abrangente.

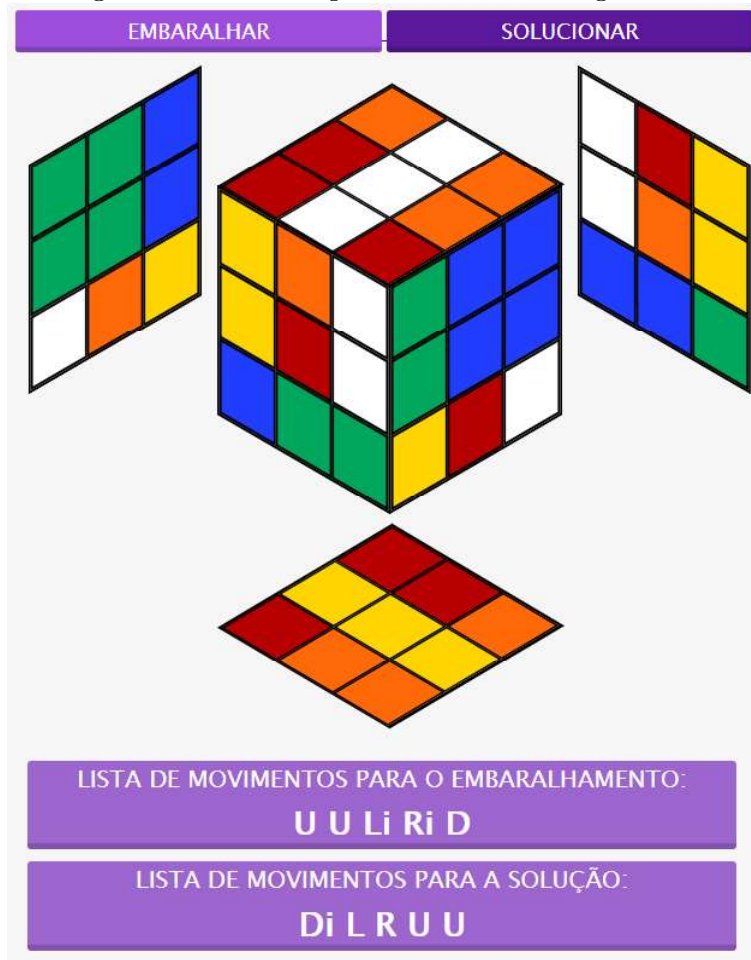


Fonte: Do autor, 2020.

### 3.3 Interface

Após escolhermos a visualização do Cubo de Rubik bidimensional abrangente, partimos para o entendimento dele dentro da interface interativa. Com isso, considerando que estará ao lado de botões clicáveis e interativos com ele, é importante que o usuário seja capaz de entender como cada movimento afeta a resolução do Cubo. De tal modo, a simples fixação da estrutura do Cubo e a atualização de seus parâmetros que representam as peças é essencial para tal objetivo.

Figura 6: Interface para o Cubo e o Algoritmo.



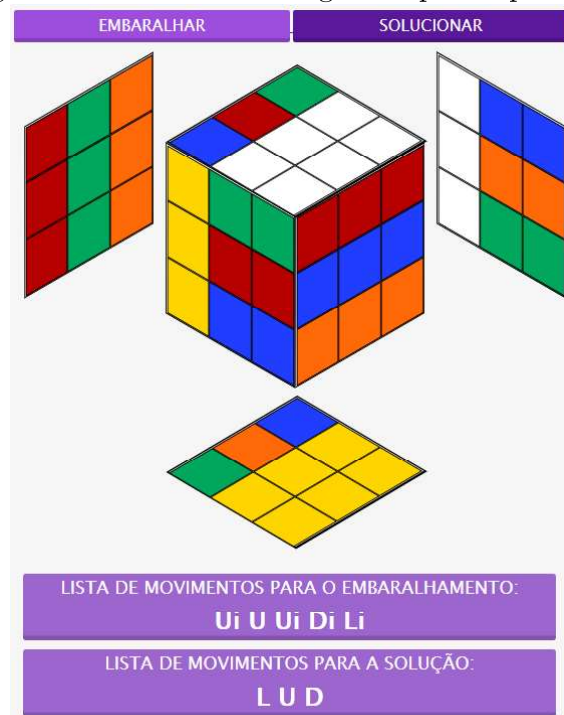
Fonte: Do autor, 2020.

Apronfundando no estudo dos métodos do Cubo Mágico, vários deles utilizam de uma tática de etapas para conclusão do Cubo, sempre focando em identificação de padrões e solucionar pequenas etapas dentro das maiores. No entanto, por se tratar de um método realizado por um algoritmo que não é de reconhecimento, a resolução pode trazer reflexões importantes para o entendimento por completo do problema.

### 3.3.1 Simulação

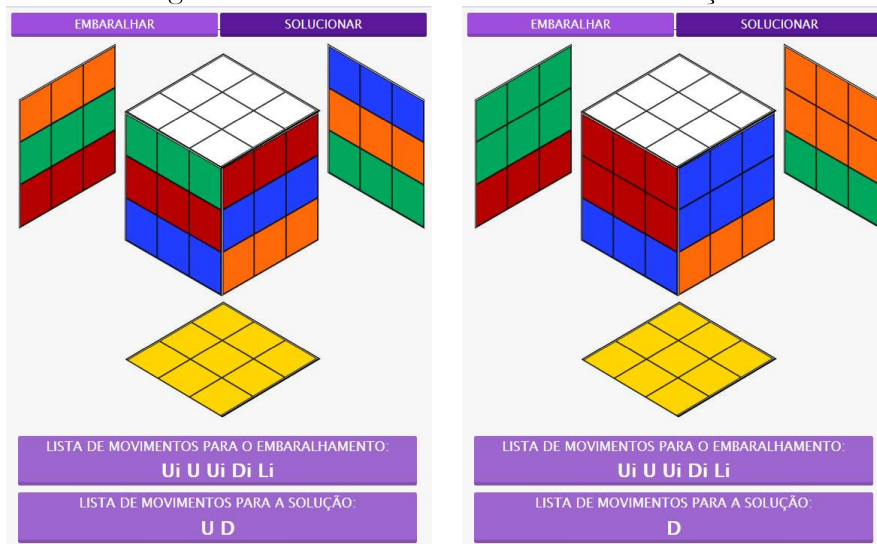
Ao entrar na aplicação, um Cubo de Rubik embaralhado é gerado aleatoriamente e tem as listas de movimentos para embaralhar e solucionar expostas. Ao clicar no botão de Solucionar, o usuário consegue visualizar os movimentos da solução do cubo acontecendo, até que seja alcançada a solução do problema.

Figura 7: Cubo de Rubik gerado pela Aplicação.



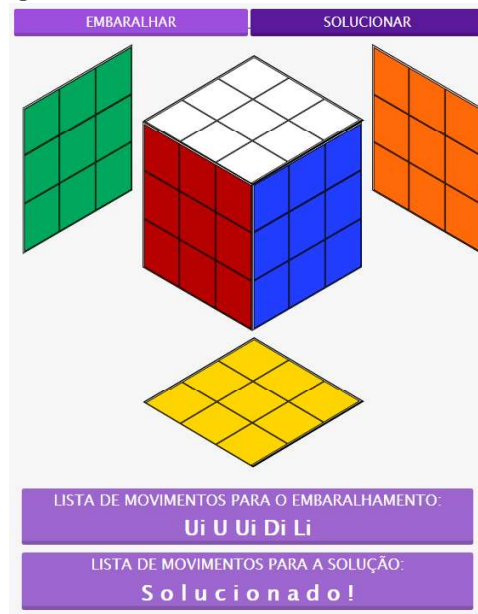
Fonte: Do autor, 2020.

Figura 8: Estados Intermediários da Solução.



Fonte: Do autor, 2020.

Figura 9: Cubo de Rubik Solucionado.



Fonte: Do autor, 2020.

## 4 Soluções

O próprio inventor do Cubo de Rubik, só conseguiu solucioná-lo após um mês de sua criação, em 1974 [5]. Esse fato se deve principalmente à complexidade do problema que ele havia montado. Por se tratar de um puzzle com quintilhões de cenários possíveis, era de se imaginar que a solução não fosse trivial. Com o objetivo de entender por completo essa situação, vamos explorar os primeiros passos dessa resolução.

Ao longo dos anos, diversos entusiastas e especialistas enfrentaram o desafio de encontrar uma solução para o problema, além de alguns que se arriscaram em busca da solução ótima. Dentre eles, destacamos Herbert Kociemba, um dos responsáveis por melhorar a solução por meio de grupamentos do Cubo. Além dele, diversos outros foram capazes de melhorar antigas propostas de soluções até chegarem ao que conhecemos hoje como "The God's Number", teorema que garante a resolução de qualquer Cubo mágico em até 20 movimentos.

A teoria dos grupamentos dentro do Cubo foi uma das mais exploradas ao longo dos anos. Buscando simplificar as dimensões do problema, utilizar tais grupos a partir dos movimentos tornava possível simplificar o problema para que fosse possível tratá-lo matematica e computacionalmente. No entanto, tal abordagem é complexa e explora grande parte da Teoria dos Grupos, sendo necessário um estudo completo por trás para seu entendimento.

Com o objetivo de entender os limites computacionais de algoritmos e de determinadas escolhas quando de frente ao problema do Cubo, esse trabalho tem como solução apresentar os resultados encontrados pelos algoritmos testados, além de explorar suas vantagens e desvantagens. Ademais, é importante ressaltar como a criação de uma interface interativa como meio de visualização dessas informações em tempo real.

### 4.1 Algoritmo A\*

Pensando em um problema computacional com estados e possíveis ações a partir de cada um, estamos lidando diretamente com um problema de en-

contrar o caminho até a solução. Reforçado pela busca de caminhos em grafos, um método capaz de nos fornecer a trajetória pelas possibilidades de maneira eficiente é essencial. Dessa maneira, um dos algoritmos mais otimizados para lidar com tal questão é o A\*<sup>6</sup>. Tal algoritmo foi estudado durante um projeto da graduação para a matéria de Estrutura de Dados e Algoritmos e, dessa forma, optamos por adaptá-lo ao problema desse trabalho.

#### 4.1.1 O Algoritmo

A modelagem de tal algoritmo foi realizada a partir de uma função principal e algumas auxiliares, além da criação de uma fila de prioridades, que era capaz de classificar cada estado do Cubo, fazendo com que a próxima escolha de estado seja a mais próxima da solução possível. Dessa maneira, o código consiste em:

Listing 4: Estrutura do Algoritmo A\*

---

```
class CubeState:
    def __init__(self, state, last_move, father):
        self.state = state
        self.last_move = last_move
        self.father = father
        self.distance = 0

    ### finding distance between current position in the maze and the exit
    def find_distance(self, target):
        for i in range(len(self.state)):
            if self.state[i] != target[i]:
                self.distance += 1

    ### generating all neighbors for the current position in the maze
    def gen_neighbors(self, queue, cubestate_obj, target):
        movement_list = ['U']
        for movement in movement_list:
            neighbor_cube = Cube3(state=self.state)
            neighbor_cube.move(movement)
            neighbor_cubestate = CubeState(neighbor_cube.state, movement,
                                           cubestate_obj)
            enqueue_neighbor(queue,
                             neighbor_cubestate,
                             cubestate_obj,
```

---

<sup>6</sup>Disponível em <<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>>. Acesso em 4 de out. de 2020.

```
movement,  
target)
```

---

Tal classe é responsável por coordenar a estrutura de cada estado, mantendo o seu estado atual, o movimento que o levou até ele, o estado anterior, e o seu valor de distância. Além disso, duas funções são utilizadas para gerar a distância e os estados vizinhos ao atual, respectivamente.

É importante ressaltar que ainda foram criadas quatro outras funções para organizar as iterações e a busca pelo resultado do algoritmo. Dessa maneira, ao rodar o processo como um todo, foi possível ter controle do que acontecia em cada iteração e confirmar o funcionamento normal do processo de resolução do Cubo.

Outro ponto relevante desse algoritmo é que ele está adaptado a modelagem antiga do Cubo, em que ele era representado por um *array* linear.

Listing 5: Função Principal do Algoritmo A\*

---

```
### auxiliar solve function that calls all other functions and return the  
    list of movements when the exit is reached  
def solve(queue, target):  
    cubestate_obj = queue.get()  
    if is_goal(cubestate_obj.state, target):  
        return unroll(maze_obj)  
    else:  
        cubestate_obj.gen_neighbors(queue, cubestate_obj, target)  
        return solve(queue, target)
```

---

#### 4.1.2 Resultados do Método

Visto que o problema lida com uma fila de prioridades com tamanho finito, mesmo que grande, além de um processo com 12 movimentos possíveis, era esperado que o resultado fosse um erro de *stack overflow*<sup>7</sup>. Dessa forma, algumas hipóteses surgiram e o problema precisou passar por algumas mudanças para que começasse a funcionar:

---

<sup>7</sup>Disponível em <[https://en.wikipedia.org/wiki/Stack\\_overflow/](https://en.wikipedia.org/wiki/Stack_overflow/)>. Acesso em 5 de out. de 2020.

- Otimizar a modelagem do Cubo para que os estados não fossem tão complexos de alternar entre si;
- Criar uma função heurística que fosse capaz de gerar um valor que representasse a realidade da distância do Cubo pro seu estado final;
- Utilizar algum método que controla a profundidade da fila.

## 4.2 Algoritmo IDA\*

Adaptado diretamente do A\*, o algoritmo IDA\*<sup>8</sup> é uma versão mais abrangente dele no que tange o controle do algoritmo. Capaz de limitar a profundidade da fila de prioridades que é usada para fazer a busca do próximo estado, ele resolve um dos problemas pontuados anteriormente. Além disso, é uma versão totalmente otimizada e que mantém a essência do método, tornando-o uma opção viável para esse trabalho.

A escolha desse algoritmo foi com inspiração em diversos trabalhos acerca do tema de solução do Cubo Mágico, juntamente com o aprofundamento do estudo no Algoritmo A\*.

Sua modelagem foi feita de maneira distinta, principalmente por estar lidando com a nova modelagem do Cubo. Como dito na seção 2, essa modelagem era benéfica para calcular a função heurística escolhida pelo IDA\*. Nesse caso, este trabalho utiliza o máximo da função Manhattan Distance 3D entre os cantos e os meios do Cubo de Rubik.

### 4.2.1 Matemática da Função Heurística

A escolha de uma boa função heurística para determinar a distância do estado atual ao final é definida matematicamente por uma equação simples:

$$f(n) = g(n) + h(n), h(n) \leq h^*(n) \forall n \quad (1)$$

---

<sup>8</sup>Disponível em <[https://en.wikipedia.org/wiki/Iterative\\_deepening\\_A\\*](https://en.wikipedia.org/wiki/Iterative_deepening_A*)>. Acesso em 18 de nov. de 2020.

Sendo  $f(n)$  a função que retorna o valor daquele estado,  $g(n)$  o custo do nó inicial até o nó atual e  $h(n)$  o custo do nó atual até o nó final. Além disso,  $h^*(n)$  é o custo ótimo para se chegar ao nó final a partir de  $n$ .

Dessa forma, foi escolhida a função de Manhattan Distance 3D, que mede a distância entre dois pontos a partir da fórmula:

$$d_1(p, q) = \|p - q\|_1 = \sum_{i=1}^n |p_i - q_i| \quad (2)$$

Em que  $(p, q)$  são vetores do tipo:  $p = (p_1, p_2, \dots, p_n)$ ,  $q = (q_1, q_2, \dots, q_n)$  e  $n$  é igual a 3.

Tal escolha foi feita tomando como inspiração o trabalho de Richard E. Korf [6].

Programando a função em Python, o resultado é esse:

Listing 6: Manhattan Distance 3D

---

```
def manhattan_distance(cube, i, z, corner):
    c1 = array[i, z]
    center = None
    for c in [1, 4, 7, 10, 13, 16]:
        if cube[i, z] == cube[c, 1]:
            center = c
            break
    if corner:
        c_t = array[center - 1, 0]
        d1 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] - c_t[2])
        c_t = array[center - 1, 2]
        d2 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] - c_t[2])
        c_t = array[center + 1, 0]
        d3 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] - c_t[2])
        c_t = array[center + 1, 2]
        d4 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] - c_t[2])
        return min(d1, d2, d3, d4)
    else:
        c_t = array[center - 1, 1]
        d1 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] - c_t[2])
```

---

```

c_t = array[center, 0]
d2 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] -
c_t[2])
c_t = array[center, 2]
d3 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] -
c_t[2])
c_t = array[center + 1, 1]
d4 = abs(c1[0] - c_t[0]) + abs(c1[1] - c_t[1]) + abs(c1[2] -
c_t[2])
return min(d1, d2, d3, d4)

```

---

### 4.2.2 Manhattan Distance 3D para o Cubo de Rubik

Aprofundando a função escolhida para ser a heurística do método IDA\*, vamos analisar como ele funciona e quais as vantagens de tê-lo escolhido. É importante considerar que todos os centros são peças fixas no nosso cenário, então as únicas peças móveis são os meios e os cantos.

Tomando esses pontos como referência, essa medida é calculada para cada canto, somada e dividida pelo número total de cantos, assim como o mesmo acontece com os meios. Ao final do processo, é escolhido o máximo entre essas medidas.

Listing 7: Máximo da Manhattan Distance 3D

---

```

### Calculate max between corners sum or edges sum
def corner_edge_sum_max(cube):
    corners = 0
    edges = 0
    for i in range(18):
        if i % 3 == 0 or i % 3 == 2:
            corners += manhattan_distance(cube, i, 0, True) +
manhattan_distance(cube, i, 2, True)
            edges += manhattan_distance(cube, i, 1, False)
        else:
            edges += manhattan_distance(cube, i, 0, False) +
manhattan_distance(cube, i, 2, False)
    return max(corners / 12, edges / 8)

```

---

### 4.2.3 O Algoritmo

A modelagem do novo algoritmo, tendo já a função heurística, é similar ao do primeiro algoritmo testado. No entanto, foi necessário recriar o método para que ele fosse adaptável aos novos paradigmas que foram criados pelas estruturas do Cubo e dos movimentos.

Vale ressaltar que diversas das funções auxiliares foram facilmente adaptadas e reutilizadas, como é o caso da fila de prioridades.

Listing 8: Funções Auxiliares

---

```
### Check if current state is goal
def is_goal(curr):
    if curr.h != 0:
        return False
    return True

### Unroll all movements to reach current state
def unroll(curr):
    movements = []
    while curr.parent is not None:
        parent = curr.parent
        movements.append(curr.move)
        curr = parent
    return movements

### Check if child is not it's own grandparent
def is_grandparent_child(child, parent):
    curr = parent.parent
    while curr is not None:
        if np.array_equal(curr.cube, child):
            return True
        curr = curr.parent
    return False

### Check if child is already on the priority queue
def is_child_in_queue(child, queue):
    for curr_item in queue.queue:
        if np.array_equal(curr_item[2].cube, child):
            return True
    return False
```

---

Ademais, também foi necessária a criação de classes auxiliares para lidar com o problema. De tal modo, é válido destacar que esses foram os

principais componentes para que o método se tornasse escalável e que fosse facilmente entendido durante o processo de criação.

#### Listing 9: Classes Auxiliares

---

```
### State class for each cube movement
class State:
    def __init__(self):
        self.cube = None
        self.g = 0
        self.h = 0
        self.parent = None
        self.move = None

### Modified priority queue for better usage
class my_priority_queue(PriorityQueue):
    def __init__(self):
        PriorityQueue.__init__(self)
        self.counter = 0

    def put(self, item, priority):
        PriorityQueue.put(self, (priority, self.counter, item))
        self.counter += 1

    def get(self, *args, **kwargs):
        _, _, item = PriorityQueue.get(self, *args, **kwargs)
        return item
```

---

Tendo todas as funções necessárias para criar a função principal, basta entender como o fluxo do algoritmo IDA\* funciona. Um dos principais pontos é o limitador estabelecido na busca da fila de prioridades utilizando a função heurística explicada anteriormente. Limitando a profundidade atingida na fila, é possível controlar o fluxo e evitar o erro de *stack overflow* que havia ocorrido anteriormente.

Otimizar a questão do código é crucial para todo o trabalho. Por lidar com um campo de possibilidades na casa dos quintilhões de movimentos, não é cabível imaginar um cenário em que todos são percorridos, por isso a busca direcionada e controlada é necessária. Finalmente, outro ponto relevante para tal algoritmo é a sua capacidade de lidar com estados e ações a partir dele, tornando simples a visualização do que ocorre ao longo do tempo.

Listing 10: Função Principal do IDA\*

---

```
### IDA* algorithm
def IDA(start):
    start.h = corner_edge_sum_max(start.cube)
    cost_limit = start.h
    nodes = 0
    queue = my_priority_queue()
    branching_factors = list()
    while True:
        minimum = None
        queue.put(start, start.h)
        while not queue.empty():
            curr = queue.get()
            if is_goal(curr):
                print('Goal Height:', curr.g)
                print("Nodes Generated:", nodes)
                print(unroll(curr))
                return
            b = 0
            nodes = nodes + 12
            for i in range(12):
                new = State()
                new.cube = np.array(curr.cube)
                new.g = curr.g + 1
                new.parent = curr
                new.move = make_move(new.cube, i + 1)
                new.h = corner_edge_sum_max(new.cube)
                if new.g + new.h > cost_limit:
                    if minimum is None or new.g + new.h < minimum:
                        minimum = new.g + new.h
                    continue
                if curr.parent is not None and
                   (is_grandparent_child(new.cube, curr) or
                    is_child_in_queue(new.cube, queue)):
                    continue
                queue.put(new, new.h)
                b = b + 1
            if b != 0:
                branching_factors.append(b)
        cost_limit = minimum
```

---

Portanto, utilizando todos esses métodos apresentados, é possível resolver o Cubo de Rubik para determinados estados. Para isso, além de determinar algumas métricas como o número de nós gerados, distância do estado do Cubo até o final, e o tempo demorado nesse processo, foi necessário criar um método de verificação.

Para as duas primeiras métricas, elas estão inseridos na própria função principal. Porém, a função de tempo e do método rodando em si foram separadas e serão apresentadas a seguir:

---

Listing 11: Funções Auxiliares para marcação do tempo

---

```
def init_timer():
    time.ctime()
    fmt = '%H:%M:%S'
    start = time.strftime(fmt)
    print("Started process")
    return fmt, start

def end_timer(fmt, start):
    time.ctime()
    end = time.strftime(fmt)
    print("Time taken(sec):", datetime.strptime(end, fmt) -
          datetime.strptime(start, fmt))
```

---

---

Listing 12: Método Principal

---

```
def main(scramble_moves=5):
    curr = State()
    scrambled_list = scramble(initial_array, scramble_moves)
    curr.cube = np.array(initial_array)
    #print(curr.cube)
    print(scrambled_list)
    IDA(curr)
```

---

É importante ressaltar nesse momento que existem limitações até mesmo para esse algoritmo otimizado. Tais situações serão abordados na próxima subseção e finalizadas na conclusão. Outrossim, suas vantagens também serão discutidas nesses próximos tópicos.

#### 4.2.4 Resultados do Método

A função principal do método consiste em gerar um Cubo no seu estado final, embaralhá-lo e, a partir desse novo estado, rodar o algoritmo IDA\* nele e gerar a sequência de movimentos necessários para retornar ao estado original. Além disso, o tempo que leva para esse método ser realizado é contado a partir do momento em que começa a rodar.

Logo nos primeiros testes realizados, é possível notar uma facilidade para o algoritmo lidar com o Cubo a uma distância de até 5 estados para o estado final, levando até 3 segundos para realizar todo esse processo. No entanto, ao aumentar esse número, o tempo gasto para processar tal método aumenta exponencialmente e se torna inviável, sendo esse o maior dos problemas com essa abordagem.

Podemos observar a seguir os resultados obtidos para certos números de movimentos ao embaralhar:

Figura 10: Embaralhando com 2 movimentos.

```
C:\Users\Matheus\Desktop\TCC\1main_project> python .\ida_star.py
Started process
Movements to Scramble: ['D', 'F']
Goal Height: 2
Nodes Generated: 60
Movements to Solve: ['Fi', 'Di']
Time taken(sec): 0:00:00
```

Fonte: Do autor, 2020.

Figura 11: Embaralhando com 4 movimentos.

```
C:\Users\Matheus\Desktop\TCC\1main_project> python .\ida_star.py
Started process
Movements to Scramble: ['R', 'Fi', 'B', 'B']
Goal Height: 4
Nodes Generated: 252
Movements to Solve: ['Bi', 'Bi', 'F', 'Ri']
Time taken(sec): 0:00:00
```

Fonte: Do autor, 2020.

Figura 12: Embaralhando com 6 movimentos.

```
C:\Users\Matheus\Desktop\TCC\1main_project> python .\ida_star.py
Started process
Movements to Scramble: ['Li', 'Ri', 'U', 'Ri', 'Bi', 'R']
Goal Height: 6
Nodes Generated: 108876
Movements to Solve: ['Ri', 'B', 'R', 'Ui', 'L', 'R']
Time taken(sec): 0:00:55
```

Fonte: Do autor, 2020.

Após observarmos os resultados obtidos, fica claro a maneira como o problema escala de modo que se torna inviável armazenar tantos estados e nós gerados. Dessa maneira, o tempo e o número de nós vão aumentando exponencialmente, inviabilizando a solução do problema para casos mais extremos.

A quantidade de movimentos de embaralhamento mais comum são entre 15 e 20 movimentos, fazendo com que o algoritmo se torne um tanto inviável a partir desse momento. Contudo, desde que a distância do estado do Cubo esteja em até 6, ainda é possível que ele seja resolvido em menos de 1 minuto.

## 5 Conclusão

Neste trabalho buscamos entender o funcionamento do Cubo de Rubik, encontrando uma maneira de modelar, visualizar e o solucionar em ambiente computacional. A partir da escolha de modelagem, foi possível criar uma interface interativa do Cubo capaz de explorar o embaralhamento e o encontro de uma solução para o problema. Ademais, a partir do estudo aprofundado de algoritmos de busca em caminhos, encontramos o IDA\*, capaz de solucionar o Cubo para determinados estados de acordo com a distância de movimentos.

As limitações encontradas na modelagem linear do Cubo foram resolvidas a partir da nova modelagem matricial. Além disso, a partir dela foi possível encontrar uma função heurística como o máximo da Manhattan Distance 3D para cantos e meios, direcionando o percurso por um algoritmo de busca em caminhos capaz de suportar a dimensão de todos os estados possíveis do Cubo.

Acerca dos algoritmos A\* e IDA\*, foi possível perceber a diferença clara de otimização entre ambos. Enquanto o A\* acumulava diversos estados do Cubo de Rubik e travava ao chegar em um *array* maior do que a capacidade do computador, o IDA\* era capaz de contornar essa situação limitando a busca da árvore de possibilidades. No entanto, mesmo sendo capaz de limitar, ainda há uma barreira para tal algoritmo, não sendo possível processar tantos estados para um Cubo a mais de seis movimentos de distância do Cubo completo.

Portanto, é evidente a capacidade que os algoritmos mais atuais tem de resolver problemas com dimensões na casa dos quintilhões. - como é o caso do Cubo de Rubik. No entanto, ainda que exista tal limitação de busca, outras abordagens podem tornar tal resolução mais eficiente e, consequentemente, viabilizar a busca com maior profundidade. Como trabalhos futuros, tais opções surgiram: utilizar uma função heurística capaz de observar um banco de dados com estados pré-existentes do Cubo e suas respectivas distâncias; modelagem de uma rede neural capaz de receber o estado de um Cubo e retornar os movimentos que direcionam para a solução.

## 6 Referências

- [1] James G. Nourse. The Simple Solution to Rubik's Cube. 1981.
- [2] Tomas Rokicki, Herbert Kociemba, Morley Davidson, John Dethridge. God's Number is 20. [2010?].  
Disponível em <<https://www.cube20.org/>>. Acesso em: 21 out. 2020.
- [3] Adrian Liaw. PyCuber. 2015.  
Disponível em <<https://github.com/adrianliaw/PyCuber>>. Acesso em 22 out. 2020.
- [4] Módus Beke. Rubik's Cube Solver. [2015?].  
Disponível em <<https://rubiks-cube-solver.com/>>. Acesso em 24 out. 2020.
- [5] QUARTZ. It took the inventor of the Rubik's Cube a month to solve his own puzzle. 2017.  
Disponível em <<https://qz.com/935952/it-took-the-inventor-of-the-rubiks-cube-a-month-to-solve-his-own-puzzle/>>. Acesso em 24 out. 2020.
- [6] Richard E. Korf. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. 1997.