Relatório Projeto

Pedro Daniel Camargos Soares **0020640**

Lucas Gabriel de Almeida **0035333**

Problema da Coloração de Grafos

Introdução

Definição de Grafo

Um grafo é uma estrutura de dados composta de vértices (ou nós) V, e arestas E. É geralmente usado para representar situações do mundo real, onde vértices representam objetos e as arestas representam relações entre eles.

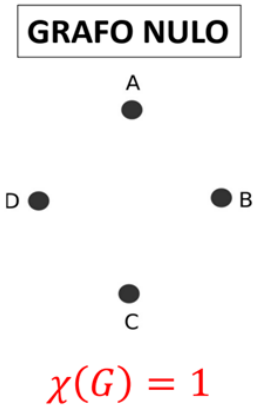
Problema de coloração

Um problema de coloração em grafos consiste em atribuir cores aos vértices ou arestas do grafo, seguindo determinadas condições. O mais comum é atribuir o menor número de cores possível aos vértices do grafo, de forma que dois vértices que possuem relação não sejam coloridos com a mesma cor.

Número Cromático

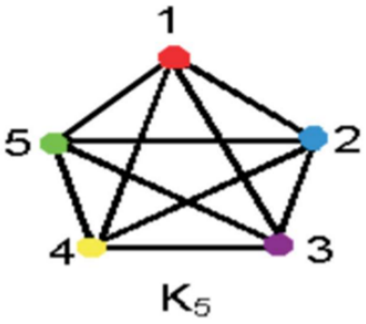
O número cromático de um grafo G representa o menor número de cores necessárias para colorir os vértices de um grafo, sem que vértices adjacentes tenham a mesma cor. É denotado por $\chi(G)$.

Exemplos de grafos com números cromáticos definidos:



Como o grafo nulo não possui nenhuma aresta, podemos colorir todo o grafo utilizando apenas uma única cor. Por tanto seu número cromático é $\chi(G) = 1$

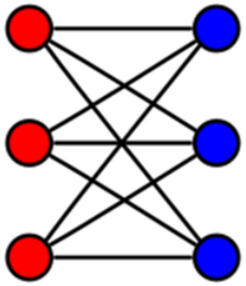
GRAFO COMPLETO



$$\chi(K_n) = n$$

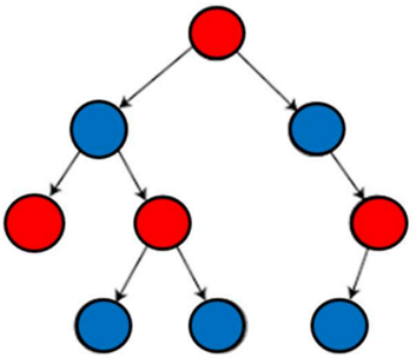
Como no grafo completo todos os vértices são adjacentes a todos os outros vértices, para cada vértice será necessário atribuirmos uma cor diferente. Por tanto seu número cromático é χ(G) = N, onde N é o número total de vértices do grafo.

GRAFO BIPARTIDO



$$\chi(G) = 2$$

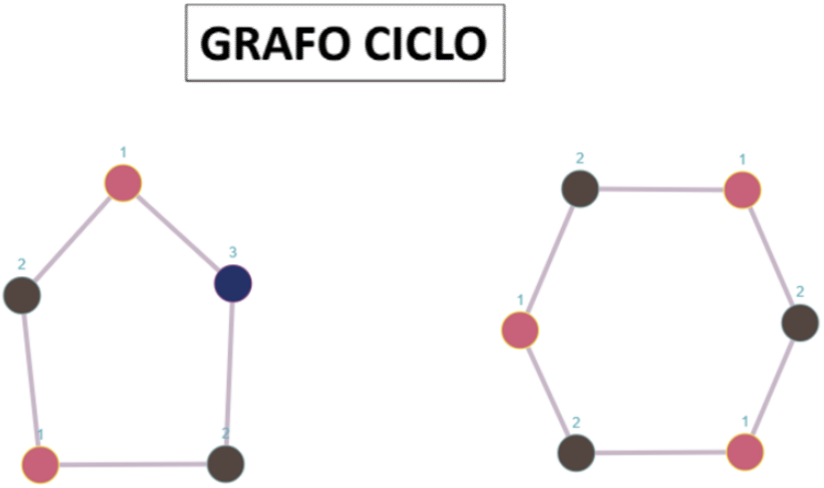
ÁRVORE



$$\chi(K_n) = 2$$

Grafos Bipartidos possuem χ(G) = 2, pois podemos separar os vértices em dois conjuntos, ondes os vértices de determinado conjunto não irão se relacionar entre si, apenas com os de outro conjunto, logo χ(G) = 2.

Toda árvore pode ser considerada um grafo bipartido, portanto elas também possuem χ(G) = 2, pois podemos utilizar uma cor para os níveis ímpares, e uma cor para os níveis pares, desta forma apenas duas cores são necessárias para colorir uma árvore.



Um grafo ciclo é aquele em que suas arestas formam um único ciclo, ligando todos os vértices. Caso o grafo ciclo contenha um número par de vértices, ele pode ser colorido com apenas duas cores, por tanto $\chi(G) = 2$. Porém se ele possuir um número impar de vértices, um dos vértices precisa ser colorida com uma cor diferente das demais, por tanto $\chi(G) = 3$.

Complexidade do problema

Como visto anteriormente, existem casos onde podemos encontrar o número cromático facilmente, porém na maioria dos casos isso não ocorre. Segundo Garey e Johnson, o problema da Coloração de Grafos é considerado NP-difícil, ou seja, ainda não existe algoritmo que consiga encontrar o menor número de cores em tempo polinomial, e a não ser que P = NP, esse algoritmo não existe.

Atualmente podemos resolver o problema através de algoritmos não polinomiais, ou através de algoritmos aproximados, que podem não encontrar a melhor solução mas garantem uma boa solução.

Origem da coloração

O problema da Coloração em Grafos surgiu em 1852, quando o matemático Francis Guthrie conjecturou que qualquer mapa político pode ser colorido com no máximo quatro cores, de forma que regiões vizinhas tenham cores distintas.

Esse teorema é considerado a prova mais “feia” da matemática, pois permaneceu sem resolução até 1976, quando os matemáticos Appel, Haken e Koch utilizaram o computador mais rápido da época para realizar bilhões de operações para provar que o teorema é verdadeiro.

Aplicações

O problema da coloração de grafos pode ser aplicado em diversas aplicações, dentre elas podemos citar:

- **Escalonamento de horários**

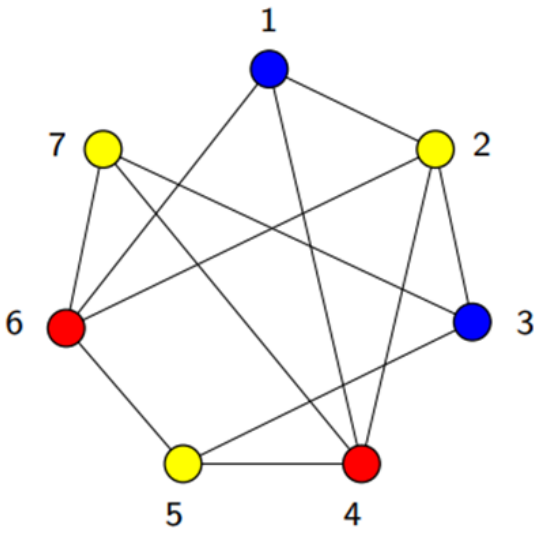
Supondo que seja necessário agendar horários de avaliações de uma universidade de modo que duas disciplinas com estudantes em comum não tenham exames agendados para o mesmo horário. Considerando que haja N disciplinas, é possível agendar as avaliações de forma que não haja conflito? A resposta é sim, basta que cada avaliação seja realizada em um dia diferente, desta forma nunca haverá conflito.

Porém esta é uma abordagem falha, já que apesar de ser bom para o aluno realizar apenas uma avaliação no dia, é desnecessário gastar vários dias para concluir todas as avaliações diferentes, visto que podemos realizar mais de uma avaliação no mesmo dia.

Portanto a pergunta muda, qual o número mínimo de dias necessários para agendar todos os exames de forma que não haja conflito de alunos?

Podemos modelar usando o problema de coloração de grafos, onde cada vértice do grafo se torna uma avaliação, e caso duas avaliações tenham conflito entre si, adicionamos uma aresta entre elas. Ao montar o grafo executamos o algoritmo de coloração e o número cromático resultante será a quantidade mínima de dias necessários para realizar as avaliações de forma a não houver conflito de alunos entre as disciplinas.

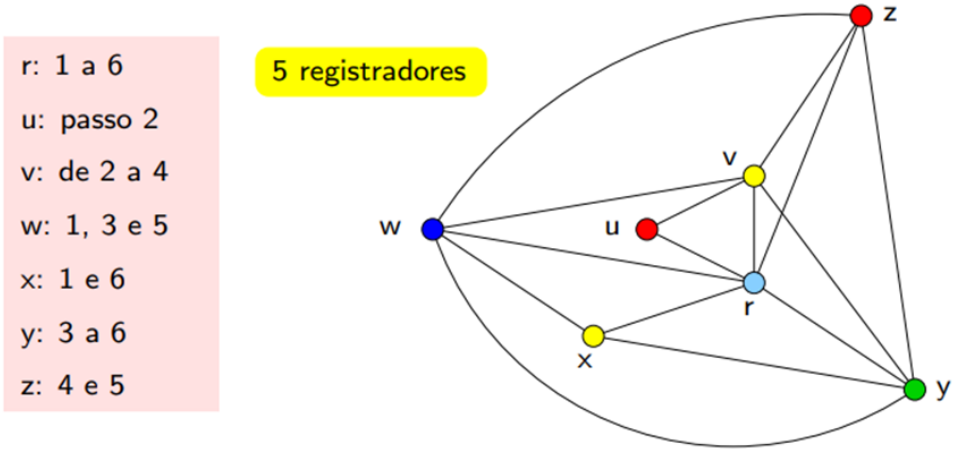
	1	2	3	4	5	6	7
1	—	X	—	X	—	X	—
2		—	X	X	—	X	—
3			—	—	X	—	X
4				—	X	—	X
5					—	X	—
6						—	X
7							—



- **Alocação de Registradores**

Sabemos que definir quais registradores serão utilizados para armazenar uma variável é uma tarefa difícil, mas é possível resolver esse problema utilizando a coloração de grafos.

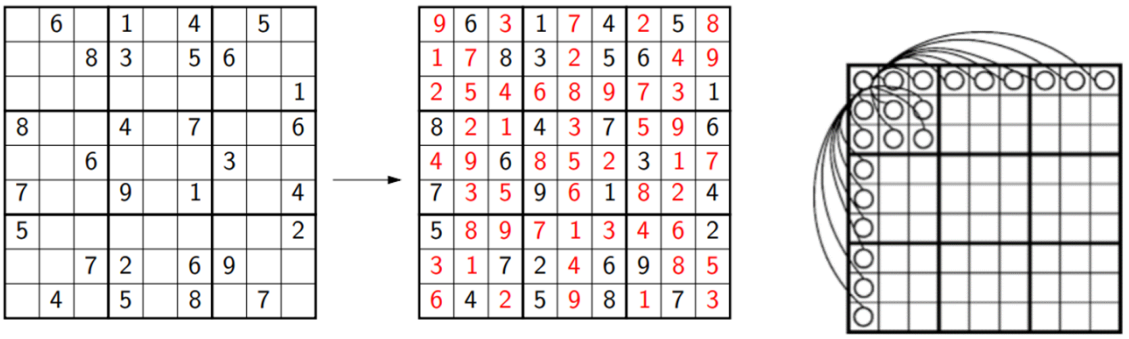
Para isso definiremos que cada vértice do grafo represente uma variável, e que adicionaremos uma aresta entre os vértices caso elas não possam ocupar o mesmo registrador, e por fim cada cor representara um registrador diferente. Ao executarmos o algoritmo de coloração de vértices, o número cromático resultante será a quantidade mínima de registradores necessárias para armazenar todas as variáveis durante a execução.



- **Sudoku**

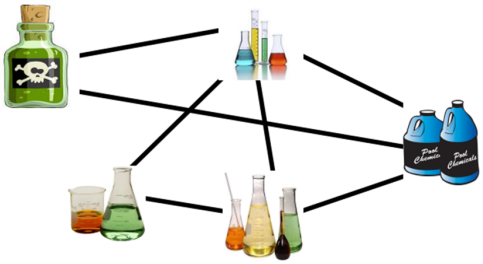
O Sudoku é um dos jogos de quebra-cabeça mais populares do mundo. O objetivo do jogo é a coloração de números de 1 a 9 em cada uma das células vazias numa grade 9x9, constituída por 3x3 subgrades chamadas regiões. A grade contém alguns números preposicionados que funcionam como dicas iniciais que oferecem uma dedução de quais números o jogador deve inserir nas células vazias. Cada coluna e linha só pode ter um dos números de 1 a 9, e o mesmo vale para as subgrades.

Podemos resolver o quebra-cabeça Sudoku utilizando a coloração de grafos, onde cada célula se torna um vértice, e existira uma aresta entre dois vértices se eles estiverem em uma mesma linha, mesma coluna ou mesma subgrade. Existirão 9 cores, e os números preposicionados (dicas) definiram a cor de alguns vértices.



- ## Grafo de Incompatibilidade

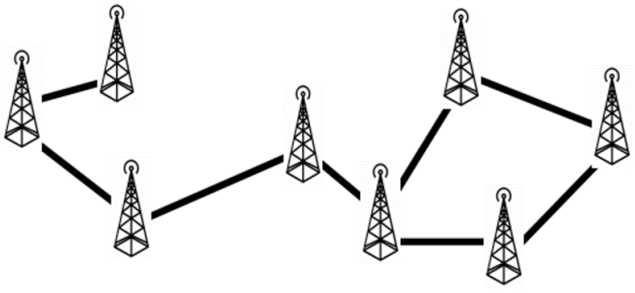
Grafos de incompatibilidade são aqueles em que dado um conjunto de N vértices, haverá uma aresta entre dois vértices caso eles não possam ter relação entre-si. Existem inúmeras aplicações de grafos de incompatibilidade na vida real, como por exemplo:



o **Armazenamento de produtos**, onde cada vértice se torna um produto, e supondo que haverá produtos que não podem ser guardados próximos a outros produtos, adicionaremos arestas entre eles. Desta forma, ao executarmos o algoritmo de coloração de vértices, encontraremos a quantidade mínima de depósitos necessária para armazenar os produtos em segurança.

o **Criação de animais**, onde cada vértice se torna uma espécie de animal diferente, e supondo que haverá animais que não podem viver no mesmo ambiente que outros animais, adicionaremos arestas entre eles. Desta forma, ao executarmos o algoritmo de coloração de vértices, encontraremos a quantidade mínima de ambientes necessária para criar os animais.

o **Alocação de frequências de rádio**, onde cada vértice se torna uma transmissão de estação de rádio diferente, e inseriremos arestas entre duas estações de rádio quando suas áreas de transmissão se sobrepuseram, o que resultaria em interferência se usassem a mesma frequência. Desta forma ao executarmos o algoritmo de coloração de vértices, encontraremos a quantidade mínima de frequências necessária para realizar as transmissões de rádio, de forma que uma estação não cause interferência a outra, e estações com a mesma cor poderão receber a mesma frequência.



Decisões de implementação que foram importantes no desenvolvimento do trabalho;

Para a implementação do trabalho foi escolhida a linguagem de programação python, na sua versão 3.10.0, bem como algumas bibliotecas que facilitaram o desenvolvimento
Foi utilizada a IDE Visual Studio Code

- Defaultdict do modulo collections: foi usada para a implementação da fila de prioridades
- Heapq, que é uma fila de prioridade usando heap
- Time, para medir o tempo de execução dos algoritmos
- Sys, para usar os argumentos de entrada do algoritmo

Ambiente computacional utilizado e a descrição dos procedimentos de testes realizados;

O programa foi desenvolvido na linguagem Python e executado no terminal integrado do visual studio code
Os testes foram executados no meu computador pessoal. Possuindo as seguintes especificações:

- Processador AMD A10-9600P RADEON R5, 10 COMPUTE CORES 4C+6G 2.40 GHz
- RAM instalada 8,00 GB (utilizável: 6,96 GB)
- Tipo de sistema Sistema operacional de 64 bits, processador baseado em x64
- AMD randeon R5 Graphics

Algoritmos

- **Guloso**

O algoritmo guloso sempre escolhe a opção que parece ser melhor naquele momento, fazendo uma escolha que será ótima local, na esperança que resulte na escolha ótima global. A cada iteração seleciona um vértice na ordem da sequência da entrada, e colore esse vértice com a cor que apresenta o menor índice possível.

Algoritmo em python

```
def coloracao_gulosa(self):
      # deixa todos os vertices com um valor de cor invalido para serem coloridos
      resultado = [-1] * int(self.vertices)
      # coloca a primeira cor no primeiro vertice
      resultado[0] = 0;
      # vetor temporario para armazenar as cores disponiveis
      # caso o valor da cor seja True, significa que ela ja foi colocada em um vertice
adjacente
      disponivel = [False] * int(self.vertices)
      #atribui uma cor aos vertices restantes do grafo
      for u in range(1, int(self.vertices)):
         # processa todos os vertices adjacentes e coloca suas cores como insisponiveis
         for vizinho, peso in self.adj[u]:
            if (resultado[vizinho] != -1):
               disponivel[resultado[vizinho]] = True

         # procura a primeira cor disponivel
         cor = 0
         while cor < int(self.vertices):
            if (disponivel[cor] == False):
               break
            cor += 1
         # então atribui a cor disponivel ao vertice em questão
         resultado[u] = cor
         # reseta os valores das cores para False para as proximas iterações
         for vizinho, peso in self.adj[u]:
            if (resultado[vizinho] != -1):
               disponivel[resultado[vizinho]] = False
      # retorna o resultado
      return resultado, max(resultado)+1
```

- **Welsh-Powell**

Esse algoritmo foi proposto pelos matemáticos Welsh e Powell em 1975. O algoritmo de Welsh-Powell é um algoritmo guloso, e nos oferece um número cromático próximo ou igual ao menor possível, tudo isso em tempo de $O(n^2)$.

O algoritmo funciona da seguinte maneira: Primeiro marcamos todos os vértices como não coloridos, em seguida montamos uma lista que contém todos os vértices e os organizamos por ordem decrescente por grau. Iniciamos um contador com o valor i=1, e atribuímos uma cor ao primeiro vértice Ci sem cor da lista. Depois percorremos o restante da lista atribuindo a mesma cor de Ci ao próximo vértice incolor não adjacente a um vértice anterior da lista de cor Ci. Atualizamos o contador e se ainda existirem vértices sem cores na lista, repetimos o processo. Ao fim imprimimos o grafo com os vértices coloridos.

Algoritmo em python

```
def coloracao_Welsh_Powell(self):
    # deixa todos os vertices com um valor de cor invalido para serem coloridos
    resultado = [-1] * int(self.vertices)
    # vetor auxiliar que guarda os vertices que ainda não possuem cores
    restante = []
    for i in range(int(self.vertices)):
        restante.append([i,len(self.adj[i])])
    # os vertices são organizados de acordo com o seu grau, do maior para o menor
    restante = sorted(restante, key=lambda x: x[1],reverse=True)
    #começa com a primeira cor
    color = 0
    #vetor auxiliar para salvar os vertices adjacentes ao vertice em questão
    coloradj = []
    # vetor auxiliar que guarda os vertices que ainda não possuem cores
    # porem, só ira armazernar os vertices ja organizados, sem o seu grau
    rest = [item[0] for item in restante]
    # enquanto existir vertices sem cores, faça
    while len(rest) != 0:
        #pega o primeiro vertice do vetor e da a ele a primeira cor disponivel
        u = rest.pop(0)
        resultado[u] = color
        # salva os vertices adjacentes ao vertice em questão
        coloradj = []
        coloradj.append(u)
        coloradj+=self.adjaux[u]
        # pega todos os vertices ainda não coloridos para que sejam iterados
        resto = []
        resto+= rest
        #variavel auxiliar para salvar o index do vertice
        index=0
        # enquanto existir vertices sem cores, faça
        while len(resto) >0:
            i = resto.pop(0)
            # caso o vertice não seja visinho de um vertice colorido com a cor em questão, seja colorido
            if (i not in coloradj):
                resultado[i] = color
                x = rest.pop(index)
                coloradj+=self.adjaux[i]
                index-=1
            index+=1
        color+=1
    #retorna o resultado e o numero de cores necessarias
    return resultado, color
```

Comparação dos Algoritmos

Implementamos ambos os algoritmos (Guloso e Welsh & Powell) na linguagem Python, e testamos suas eficiência em duas benchmarks diferentes: A benchmark disponibilizada no classroom para a geração de Árvores Geradoras Mínimas, composta de 5 grafos diferentes, e a benchmark disponibilizada pelo google, de onde escolhemos arbitrariamente 3 grafos para realizar os testes

Detalhe dos Grafos (Classroom)

Grafo	Vértices	Arestas
Pequeno	8	16
Medio1	250	1273
Medio2	1000	8433
Medio3	10000	61731
Grande	1000000	7586063

Número de Cores Necessárias

Grafo	Guloso	Wp
Pequeno	4	4
Medio1	13	11
Medio2	18	16
Medio3	16	15
Grande	21	-

Tempo Médio de execução (Em segundos)

Algoritmo\Grafo	Pequeno	Médio1	Médio2	Médio3	Grande
Guloso	0,0001996	0,005705786	0,016226292	0,093297672	22,3174984
Welsh & Powell	0,0002	0,0042	0,2229926	21,7928354	-

Algoritmo Guloso

Execução\Grafo	Pequeno	Médio1	Médio2	Médio3	Grande
1	0,000998	0,01999855	0,016122341	0,093025208	14,763209
2	0	0,002142429	0,016004086	0,086020947	29,904291
3	0	0,002144575	0,017126322	0,09514308	19,469258
4	0	0,002121925	0,016000509	0,095147848	25,374485
5	0	0,002121449	0,015878201	0,097151279	22,076249
Média	0,0001996	0,005705786	0,016226292	0,093297672	22,3174984

Algoritmo de Welsh & Powell

Execução\Grafo	Pequeno	Médio1	Médio2	Médio3	Grande
1	0,001	0,005	0,085019	26,460874	-
2	0	0,003	0,093138	20,663036	-
3	0	0,005	0,781457	20,571523	-
4	0	0,004	0,079456	20,697221	-
5	0	0,004	0,075893	20,571523	-
Média	0,0002	0,0042	0,2229926	21,7928354	-

Detalhe dos Grafos (Google)

Grafo	Vértices	Arestas
huck	74	602
queen7_7	49	952
DSJC1000	1000	49629

Número de Cores Necessárias

Grafo	Guloso	Wp	Ideal
huck	11	11	11
queen7_7	10	12	7
DSJC1000	31	29	20

Tempo Médio de execução (Em segundos)

Algoritmo\Grafo	huck	queen7_7	DSJC1000
Guloso	0,000998	0,00136	0,058562
Welsh & Powell	0,00078	0,0005602	0,46832

Algoritmo Guloso

Execução\Grafo	huck	queen7_7	DSJC1000
1	0,0001	0,002	0,05
2	0,001	0,002	0,065
3	0,001	0,0009	0,06001
4	0,00099	0,001	0,0488
5	0,002	0,0009	0,069
Média	0,000998	0,00136	0,058562

Algoritmo de Welsh & Powell

Execução\Grafo	huck	queen7_7	DSJC1000
1	0,001	0	0,580275
2	0,0009	0,0009	0,602151
3	0,001	0,0009	0,464142
4	0	0	0,361986
5	0,001	0,001001	0,333046
Média	0,00078	0,0005602	0,46832

Conclusão

Com base nos resultados apresentados, podemos observar que a nossa implementação do algoritmo Guloso é extremamente mais rápida que a implementação de Welsh & Powell, porém o resultado do algoritmo de Welsh & Powell se mostrou mais preciso que o resultado do Guloso.

Nota-se também que ambas as nossas implementações podem não apresentar o resultado ideal, visto que nas benchmarks disponibilizadas pelo google, o resultado da nossa implementação do Algoritmo Guloso e do Algoritmo de Welsh & Powell apesar de apresentar um bom resultado (próximo do ideal) não apresentaram o resultado exato.

Concluímos ressaltando a importância dos algoritmos de coloração em grafos, pois além de se tratar de um problema NP-Completo, com eles é possível modelar e resolver inúmeros problemas da vida real.

Referências

LEVADA, Alexandre. ***O algoritmo Welsh & Powell***. Youtube, 20 de jan. de 2021. Disponível em: <https://www.youtube.com/watch?v=dnhXVRC0LQQ>. Acesso em: 28/12/2021

MAIOLI, Douglas. ***Coloração de Grafos - Aula 13 de Teoria dos Grafos***. Youtube, 13 de jan. de 2021. Disponível em: <https://www.youtube.com/watch?v=ZrkL_uEiiyU>. Acesso em: 28/12/2021

Graph Coloring Benchmarks. Disponível em <https://sites.google.com/site/graphcoloring/vertex-coloring>. Acesso em: 28/12/2021

LUIZ, Atílio Gomes. ***Coloração de grafos e suas aplicações.*** Universidade Federal do Ceará - Campus Quixadá. 13 de maio de 2015. Disponível em: <https://www.ic.unicamp.br/~atilio/slidesWtisc.pdf>. Acesso em: 28/12/2021

Pardo, Thiago A. S; Oliveira, Maria Cristina F. ***Coloração de grafos.*** Disponível em: <https://edisciplinas.usp.br/pluginfile.php/5262366/mod_resource/content/2/6.%20Colora%C3%A7%C3%A3o%20de%20grafos.pdf>. Acesso em: 20/01/2022

<https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/>. Acesso em: 20/01/2022