Coloração Aproximada de Grafo com Algoritmos Guloso e Backtracking

Discente: Anny Caroline Walker Silva | Docente: Prof. Dr. Herbert Rocha

***Resumo*—Nesse trabalho é apresentada uma introdução sobre os conceitos das classes de problemas de complexidade computacional NP, NP-Difícil e NP-Completo e uma prova de verificação de que o problema de Coloração de Grafo pertence a classe NP-Completo, assim como implementação de um Algoritmo Guloso e Algoritmo de Backtracking para coloração de grafo, que no final são comparados e avaliados.**

***Palavras-chave*—Coloração de Grafo, 3-COLOURING, NP-Completo, Algoritmo Guloso, Algoritmo Backtracking**

I. INTRODUÇÃO

O problema da coloração de grafo é um problema de otimização combinatória, consiste em dado um grafo G (V, E), onde V é conjunto de vértices e E o conjunto de arestas, deve-se colorir cada vértice em V de forma que os vértices vizinhos tenham cores diferentes (MANN e SZAJKÓ, 2013).

Apenas a coloração dos vertices do grafo pode ser um problema formalizado através da seguinte expressão: ∀ (u,v) ∈ E: c(u) ≠c(v), ou seja, para cada dupla u e v de vertices adjacentes deve-se ter uma cor c diferente.

Uma especificação mais detalhada desse problema é a otimização de cores, onde temos no máximo 3 cores para distribuir pelos vertices do grafo sem repetição em seus adjancentes. Essa restrição de coloração de grafo resulta em um problema NP-Completo (MAVATADID, 2014).

Problemas NP-Completo são uma classe de problemas da área de complexidade computacional que podem ser verificados em tempo polinomial, ou seja, são solucionáveis com tempo de execução de limite superior expresso por polinômios a partir do tamanho da entrada. Formalmente descrito como T(n) = O (n^k), onde n é o tamanho da entrada do problema T, O é o limite superior de tempo de execução para verificar esse problema e k é uma constante.

Dentre as várias alternativas para verificação da Coloração Aproximada de Grafos, temos o algoritmo guloso e o algoritmo backtracking. O algoritmo guloso consiste na estratégia de procurar a solução de um determinado problema a partir da escolha do melhor caminho a partir do momento atual da verificação, que inicia na primeira entrada, até se satisfazer com um parâmetro de parade.

Enquanto o algoritmo backtracking aplica a estratégia da força bruta, avaliando possíveis candidatos para a solução do problema a partir de uma busca em profundidade até encontrar o ultimo elemento da entrada, inicia-se o processo de volta a origem.

II. COLORAÇÃO DE GRAFOS COMO UM NP-COMPLETO

A classe de problemas NP-Completo fica entre a interseção dos conjuntos de problemas NP-Difícil e NP. Onde NP é um classe de problemas que podem ser resolvidos e verificados com tempo polinomial não determinístico a partir de algoritmos não determinísticos, que são algoritmos que encontram a solução exata e ótima em tempo polinomial.

Já os problemas do tipo NP-Difícil são problemas que são pelo menos tão difíceis quanto os problemas mais difíceis da classe NP. Na Figura 1 é apresentado um diagrama dos conjuntos dos problemas NP, NP-Difícil, NP-Completo e P.

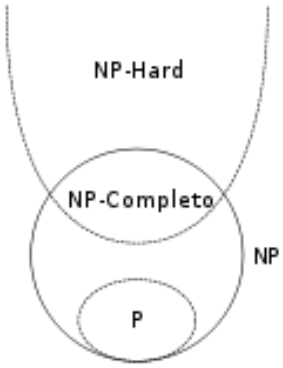

Figura. 1. Diagrama de problemas P, NP, NP-Difícil e NP-Completo.

Um problema é dito como NP-Completo se é verificado que está em NP, e se todo problema em NP-Difícil é redutível em tempo polinomial.

A coloração de grafo com o mínimo uso de cores k possível pertence a classe de problemas NP-Completo, quando k≤3, chamaremos esse problema de 3-COLOURING. A prova a seguir é um resumo do trabalho de Mavatadid (2014), que primeiramente prova que esse problema pertence a classe NP e depois a classe NP-Difícil. A definição formal da coloração de grafo com 3 cores é:

Tabela 1. Definição do problema 3-COLOURING.

Dado um grafo G (V, E), retorne 1 se e se somente se existir uma combinação de coloração de G usando no máximo 3 cores.

Para verificar que o problema é da classe NP basta usar um

verificador com complexidade de tempo igual a O (n^2), que cria uma cor nova para cada vértice de forma a ele não ter a mesma cor de seus adjacentes, essa mesma cor pode ser replicada em outros vértices, pois antes de criação da cor é realizado uma busca de cor de menor índice possível seguindo as restrições de não repetição dos adjacentes.

Para provar que o problema é NP-Difícil deve-se executar uma redução polimonial de 3-SAT em 3-COLOURING, onde 3-SAT é o problema base da classe NP-Completo, e consiste no problema da satisfatibilidade booleana (SAT), onde se procura verificar se existe uma determinada atribuição de valores para variáveis de uma fórmula booleana, de tal forma que esses valores satisfaçam a formula, no caso o 3-SAT restringe essa valoração para no máximo 3 valores.

Seja φ uma instância de 3-SAT e $C_1, C_2, \dots, C_m$ as cláusulas de φ definido a partir das variáveis $\{x_1, x_2, \dots, x_n\}$. O gráfico G (V, E) de 3-COLOURING análogo a instância φ deve seguir duas restrições: de alguma forma deve estabelecer valores para as variáveis $x_1, x_2, \dots, x_n$ através das cores de G e capturar a satisfatibilidade de cada clausula C_i em φ.

Para fazer comparação e equivalência entre esses dois problemas, primeiro é instanciado um grafo G com 3 vértices {T, F, B} onde T é TRUE (verdadeiro), F é FALSE (falso) e B é BASE (base). Esses vértices devem equivaler ao mesmo tempo a paleta de cores para coloração do grafo 3-COLOURING e satisfação de valoração de 3-SAT.

O próximo passo é adicionar um par de vértices v_i e v'_i para cada valoração literal x_i possível de 3-SAT e criar um triângulo entre B e cada par criado. Resultando na Figura 2 abaixo.

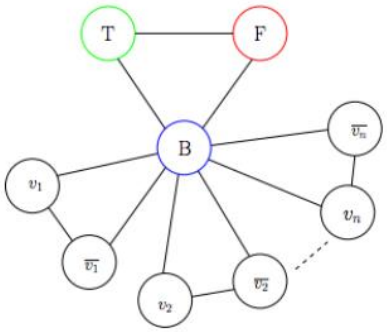

Figura. 2. Grafo de equivalência de valoração entre 3-SAT e 3-COLOURING.

Assumindo que o grafo G é 3-COLOURING, então podemos garantir que v_i ou v'_i (mas nunca os dois ao mesmo tempo) recebem a cor T. Agora que temos a valoração desejada, devemos capturar a equivalência entre as satisfatibilidades de 3-SAT e 3-COLOURING.

Para isso é usado o gadget lógico OR (ou), onde em uma cláusula C_i = {a ∨ b ∨ c} de 3-SAT precisamos expressar através do OR de seus literais usando as cores {T, F, B} do 3-COLOURING. O gráfico da Figura 3 abaixo ilustra como é alcançado a conexão entre os literais de C_i.

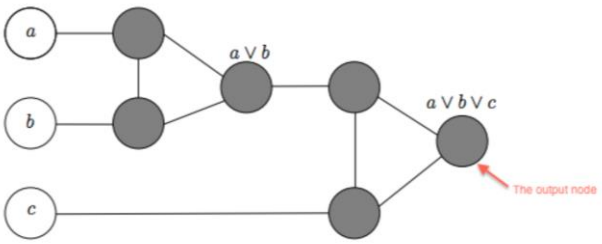

Figura. 3. Gráfico de conexão entre os literais de C_i.

O nó que captura a saída de a ∨ b é combinado com o nó que captura a saída de a ∨ b ∨ c, garantindo que se C_i é satisfeita, a coloração é dada como T para esse vértice, senão é dada como F.

Supondo que φ de 3-SAT é satisfazível e $\{x'_1, x'_2, \dots, x'_n\}$ é a valoração de satisfatibilidade, se a partir do gadget-OR x'_1 recebe TRUE, colorimos o par de vértices v_i e v'_i respecticamente com T e F, onde o terceiro vértice dessa triangularização é a base B, colorido com B, coloração essa que segue as restrições de 3-COLOURING. A Figura 4 abaixo representa a equivalência entre os problemas 3-SAT e 3-COLOURING.

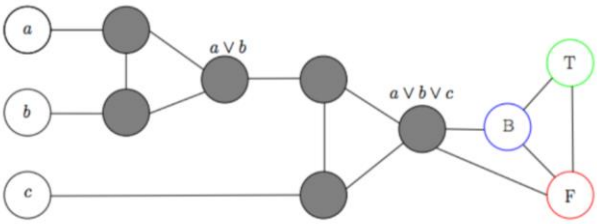

Figura. 4. Equivalência dos problemas 3-SAT e 3-COLOURING a partir da redução polinomial com uso de gadget-OR.

III. Algoritmo Guloso para Coloração de Grafo

Nesse trabalho é proposto a verificação de coloração aproximada de grafo com uso de algoritmo guloso.

A. *Descrição*

Esse algoritmo tenta resolver problemas a partir da estratégia de estágios, onde durante cada estágio é feita uma decisão localmente ótima, que ao chegar no estágio final proporciona a soma total de decisões localmente ótimas, produzindo uma solução para o problema (EBERT, 2019).

Nem sempre a soma total das decisões do algoritmo guloso resulta na solução ótima para o problema, então essa solução é do tipo aproximada, ou heurística.

A vantagem desse algoritmo é que sua implementação é simples e mesmo assim oferece uma solução aproximada, que pode ser útil em casos de problema que a exatidão da solução é dispensável.

B. *Complexidade de Tempo*

No caso do problema de coloração de grafo, a implementação de uma solução aproximada com uso do algoritmo guloso tem como pior caso a situação em que o algoritmo deve visitar todas

as arestas e vértices, onde O ($V^2 + E$), onde V corresponde ao total de vértices e E o total de arestas de um grafo G.

C. Pseudo-Código

O pseudo-código que representa o algoritmo da coloração de grafo é representado abaixo. A estratégia é primeiramente designer a primeira cor para o primeiro vértice e a partir dele colorir os próximos vértices designando cores de menor índice possível de tal forma que os adjancetes desse vértice não repitam essa cor.

Tabela 2. Pseudo-código do algoritmo guloso aplicado em coloração aproximada de grafo.

Entrada: G = (V, E)
Saída: G colorido sem vértices adjancentes com mesma cor

Início
    Seja A := $\{v_1, v_2, \dots, v_n\}$ uma sequência de vértices V de G.
    Seja C := $\{v_1, v_2, \dots, c_n\}$ uma sequência de cores.

    Cor de A[v_1] := C[1]

    Para 2 **até** n **faça**
        Processa todas as cores usadas pelos adjacentes de A[v_n]
        Procura a primeira cor em C que não seja usada pelos adjacentes de A[v_n]
    Fim Para
Fim

D. Implementação em C++

Para implementação em C++ foi definida a seguinte classe representando um grafo.

Tabela 3. Estrutura de uma classe grafo em C++.

```
class Graph {
    int V;
    list<int> *adj;

    Graph (int V) {
            this->V = V;
            adj = new list<int>[V];
     }

    ~Graph () {
            delete [] adj;
     }

    void addEdge(int v, int w);

    void greedyColoring();
};
```

Onde V comporta o número de vértices e addEdge(int v, int w) adiciona uma nova aresta recebendo como parâmetro v e w, que representam os vértices adjacentes a partir dele. E o greedyColoring() é o método que realiza a coloração, demonstrado a seguir.

Tabela 4. Função implementada em C++ que realiza coloração gulosa em grafo.

```
1   void Graph::greedyColoring()  {
2       int result[V];
3
4       result[0]  = 0;
5
6       for (int u = 1; u < V; u++)
7          result[u] = -1;
8
9       bool available[V];
10      for (int cr = 0; cr < V; cr++) {
11         available[cr] = false;
12      }
13
14      for (int u = 1; u < V; u++){
15         list<int>::iterator i;
16         for (i = adj[u].begin(); i != adj[u].end(); ++i)
17            if (result[*i] != -1)
18               available[result[*i]] = true;
19
20         int cr;
21         for (cr = 0; cr < V; cr++)
22            if (available[cr] == false)
23               break;
24
25         result[u] = cr;
26
27         for (i = adj[u].begin(); i != adj[u].end(); ++i)
28            if (result[*i] != -1)
29               available[result[*i]] = false;
30      }
31  }
```

Na linha 4 temos o primeiro vértice recebendo o índice da primeira cor, o 0. Na linha 6 podemos ver a preparação do vetor auxiliar result, que tem todo os vértices do grafo inicializados como -1, pois esse índice representa a ausência de cor.

Nas linhas 9 e 10 temos a preparação do vetor auxiliar *avaiable* que informa se uma determinada cor está ou não disponível, sendo inicializado como falso, pois a priori nenhuma cor além do 0 foi usada.

Na linha 14 podemos ver o início do loop que percorre todos os vértices do grafo. Na linha 16 são percorridos todos os adjacentes do vértice atual, e é verificado quais cores eles estão usando.

Na linha 21, 22 e 25 temos a designação da cor para o vértice atual a partir de um loop que verifica qual o primeiro índice de cor disponível no vetor auxiliar *available.*

E no vetor 27 acontece uma limpa do vetor auxiliar *available*, pois na verificação do vértice seguinte o status de cores mudaram.

E. Resultados

Para analisar a performance e tempo da implementação de algoritmo guloso sugerida no tópico anterior aplicado a coloração aproximada de grafos, foi utilizado um benchmark chamado COLOR02/03/04, mantido por Michael Trick com a colaboração de diversos pesquisadores no site mat.gsia.cmu.edu/COLOR04/.

Foram selecionadas 11 instâncias de grafos e a análise da performance da heurística gulosa foi aplicada a partir da medição do tempo de execução de acordo com o número de arestas da entrada, usando a biblioteca chrono do C++;

Tabela 5. Resultados dos testes com algoritmo guloso em milissegundo.

ID	Entrada G (V, E)	Tempo (ms)
1	ESPAÇADO - G (450, 5.714)	~0
2	ALEATÓRIO – G (905, 43.081)	2 ~ 4
3	4-FULLSINS OR ORDER 5 – G (4.146, 77.305)	5 ~ 6
4	ALEATÓRIO – G (1.809, 103.368)	7 ~ 9
5	ALEATÓRIO – G (1.870, 104.176)	7 ~ 10
6	ALEATÓRIO – G (.1870, 104.176)	7 ~ 10
7	ALEATÓRIO – G (2.368, 110.871)	8 ~ 10
8	LATIN-SQUARE 60 – G (3.600, 212.400)	16 ~ 18
9	LATIN-SQUARE 10 – G (900, 307.350)	26 ~ 30
10	ALEATÓRIO – G (4.730, 286.722)	33 ~ 38
11	ALEATÓRIO – G (1.000, 449.449)	MEMORY EXCEPTION

Através da Tabela 5 podemos analisar o crescimento do tempo de execução do algoritmo guloso de acordo com o crescimento do número de vértices e a arestas.

O algoritmo implementado não conseguiu analisar o grafo do item de ID 11, possuidor de 449449 arestas, o resultado foi um Memory Exception, investigando as possíveis motivações para esse problema, temos como primeira hipótese temos o estouro da variável que armazena o índice dos vértices, que seguindo a complexidade O ($V^2 + E$) e aplicando o V e E do ID 11, teríamos um total de 1.449.449, mas pela documentação do C++ o tipo inteiro suporta como valor máximo 2.147.483.647, limite muito superior ao índice alcançado pelo ID 11, provando que essa não é a causa do problema

A segunda hipótese é uma limitação do hardware do computador e do Windows 10, plataforma onde foram executados os testes. Novamente segundo a documentação do C++ não existe limite para vetores (estrutura que usados para armazenar os vértices do grafo), mas devido a limitação de hardwares é recomandada a alocação previa de memória para esse tipo de estrutura. Uma sugestão para uma melhoria da implementação do algoritmo guloso aqui implemetado é o uso de malloc para garantir blocos de memória para nosso programa.

IV. ALGORITMO BACKTRACKING

Nesse trabalho é proposto a verificação de coloração aproximada de grafo com uso de algoritmo backtracing.

A. Descrição

Esse algoritmo tenta resolver problemas a partir da estratégia de busca em profundidade, que utiliza o recurso de recursão para retornar depois de chegar em uma extremidade da entrada, e assim buscar por outro caminho, até que a condição de satisfação seja encontrada e o melhor caminho verificado.

Para o problema de coloração de grafos o algoritmo do backtracking procura uma solução a partir de um número m de cores onde existe uma solução que siga a restrição de não repetição de cor em adjacentes.

.

B. Complexidade de Tempo

No caso do problema de coloração de grafo, a implementação de uma solução aproximada com uso do algoritmo backtracking tem como pior caso a situação em que o algoritmo deve visitar todos os vértices a partir de um vértice para escolher a melhor combinação de cores, ou seja, a que use menor quantidade de cores, sendo O (2^V).

C. Pseudo-Código

O pseudo-código que representa o algoritmo da coloração de grafo mínimo combinação de cores é representado abaixo. A estratégia é combinar todas as cores disponíveis em um determinado vértice em seus adjacentes, e nos adjacentes de seus adjacentes de forma que use a menor quantidade de cores possível.

Tabela 6. Pseudo código de algoritmo backtracking aplicado em coloração aproximada de grafo.

Entrada: G = (V, E)
Saída: G colorido sem vértices adjancentes com mesma cor

Início
Seja A := $\{v_1, v_2, \dots, v_n\}$ uma sequência de vértices V de G.
Seja C := $\{v_1, v_2, \dots, c_m\}$ uma sequência de cores.

Para 0 **até** n **faça**
Recursão
Fim Para

Recursão
Para 1 até m faça
Se todo vértice possuir cor então
Saia da recursão

```
                    Senão
                        Checa se a cor designada disponível e
                        designa ela para o vértice
                        Recursão
                Fimpara
Fim
```

D. Implementação em C++

Para implementação em C++ foi definida a seguinte estrutura para representar um grafo.

Tabela 7. Estrutura de dado grafo em matriz;

```
bool graph[V][V];
int m;
```

Onde V comporta o número de vértices e isso gera uma matriz de adjacência entre os vértices, onde 0 indica que não são adjacentes e 1 indica que são. O inteiro m representa a quantidade máxima de cores disponível, fator determinante para saber se o algoritmo backtracking encontrará solução ou não, pois um determinado m pode resultar em falha, e disponibilizando mais cores, aumentando o m, pode-se encontrar a resposta.

A seguir o algoritmo de coloração e métodos auxiliares.

Tabela 8. Implementação em C++ do algoritmo backtracking para coloração de grafo.

```
1   bool isSafe (int v, bool graph[V][V], int color[], int c) {
2       for (int i = 0; i < V; i++)
3           if (graph[v][i] && c == color[i])
4               return false;
5       return true;
6   }
7   bool graphColoringUtil(bool graph[V][V], int m, int color[], int v) {
8         return true */
9       if (v == V)
10          return true;
11      for (int c = 1; c <= m; c++) {
12          if (isSafe(v, graph, color, c)) {
13              color[v] = c;
14              if (graphColoringUtil (graph, m, color, v+1) == true)
15                  return true;
16              color[v] = 0;
17          }
18      }
19      return false;
20  }
21
22  bool graphColoring(bool graph[V][V], int m) {
23      int color[V];
24      for (int i = 0; i < V; i++)
25          color[i] = 0;
26
27      if (graphColoringUtil(graph, m, color, 0) == false) {
28          printf("Solução não existe para esse grafo.");
29          return false;
30      }
31      printSolution(color);
32      return true;
33  }
```

Na linha 25 temos o início da coloração, e na linha 27 inicia o loop, que começa designando a cor de índice 0, ou seja, ainda não designado, no vetor auxiliar color. Na linha 30 é chamda a função recursiva graphColoring(bool graph[V][V], int m, int color[], int v), que ao retornar false significa que não existe solução para essa coloração, e ao retornar true continua na recursividade (linha 16), até que todas os vértices tenham cores não repetidas com seus adjacentes, como é descrito no caso base da linha 11;

Na linha 1 temos a função auxiliar isSafe(int v, bool graph[V][V], int color[], int,c), que verifica a partir do vértice atual se a combinação de cores testada segue as restrições do problema, se não segue, ele volta para a linha 13 e tenta uma nova combinação, que se não for possível retorna false.

Na linha 8

Nas linhas 9 e 10 temos a preparação do vetor auxiliar *avaiable* que informa se uma determinada cor está ou não disponível, sendo inicializado como falso, pois a priori nenhuma cor além do 0 foi usada.

Na linha 14 podemos ver o início do loop que percorre todos os vértices do grafo. Na linha 16 são percorridos todos os adjacentes do vértice atual, e é verificado quais cores eles estão usando.

Na linha 21, 22 e 25 temos a designação da cor para o vértice atual a partir de um loop que verifica qual o primeiro índice de cor disponível no vetor auxiliar *available.*

E no vetor 27 acontece uma limpa do vetor auxiliar *available*, pois na verificação do vértice seguinte o status de cores mudaram.

E. Resultados

Essa instância do algoritmo backtracking tem limitações com relação a entrada do grafo, pois é reconhecido e montado grafo a partir de matriz de adjacência e não foi encontrado um benchmark adequado que oferecesse matrizes nesse padrão.

Dessa forma foi utilizado um gerador de matrizes, que possibilita a geração de matrizes aleatórias com valores 0 e 1 a partir de uma entrada para tamanho de coluna e linha, no site onlinemathtools.com/generate-random-matrix.

Foram geradas matrizes aleatórias de 5, 10, 25, 50, 100. O problema é que não há como garantir que não existem ciclos nessas matrizes, o que resultou nos seguintes dados:

Tabela 9. Resultados da aplicação do algoritmo backtracking de acordo com as matrizes[V][V].

ID	Tamanho da matriz[V][V]	Quantidade de cores	Tempo (ms)
1	matriz[5][5]	m=2	NaN
		m=3	11~74
2	matriz[10][10]	m=3	NaN
		m=4	16~87
3	matriz[25][25]	m=5	NaN
		m=6	25~93
4	matriz[50][50]	m=5	NaN (228)
		m=6	NaN (31961)
5	matriz[100][100]	m=5	NaN (533)
		m=6	NaN (143843)
		m=7	-

NaN representa que dado o grafo da matriz[V][], com a quantidade m de cores não é possível encontrar solução que obedeça a restrição de adjacentes com cores diferentes. O caractere "-" representa que a decisão entre é possível e não possível nunca foi encontrada, pois a execução não chegou a terminar.

Percebemos pela limitação da análise com relação ao número de entradas que a implementação desse algoritmo como é apresentando na Tabela 8 é ineficiente para entradas grandes, tanto pela sua complexidade altíssima que é O (2^V).

V. Comparação entre Algoritmo Guloso e Algoritmo Backtracking e Conclusão

De acordo com os resultados da aplicação do algoritmo guloso e algoritmo backtracking verificamos que existem vantagens e desvantagens da escolha de uso de cada um. Na Tabela 10 a diferença é gritante com relação ao tempo e quantidade de vértices, onde o backtracking implementado aparenta ser bem ineficiente.

Tabela 10. Resultado de tempo e tamanho de entrada dos algoritmos guloso e backtracking lado a lado.

GULOSO		BACKTRACKING	
Tempo (ms)	G (V, E)	Matriz[V][V]	Tempo (ms)
~0	G (450, 5.714)	matriz[5][5]	11~74
2 ~ 4	G (905, 43.081)	matriz[10][10]	16~87
5 ~ 6	G (4.146, 77.305)	matriz[25][25]	25~93
7 ~ 9	G (1.809, 103.368)	matriz[50][50]	NaN (31961)
7 ~ 10	G (1.870, 104.176)	matriz[100] [100]	NaN (143843) -

O backtracking a partir da força bruta, que verifica todas as possibilidades de combinações possíveis, retorna um resultado mais exato, porém sua alta complexidade torna a implementação para grandes entradas mais complexa, deve ser feito com cuidado, escolhendo bem o tipo de entrada do grafo.

Quanto ao algoritmo guloso, existe a vantagem de sua fácil implementação e complexidade retornar um resultado aproximado útil de acordo com a criticidade de importância do fator exatidão, que é também sua desvantagem, pois oferece uma heurística do que pode ser a solução do problema.

VI. Referências

MAVATADID, Lalla. Introduction to Complexity Theory: 3-Colouring is NP-complete, 2014. Disponível em: <http://cs.bme.hu/thalg/3sat-to-3col.pdf>.

MICROSOFT. Referência de C++ bibliotecas C/Language e Standard, 2019. Disponível em: <https://docs.microsoft.com/pt-br/cpp/cpp/c-cpp-language-and-standard-libraries?view=vs-2019>.

ALPHA, Wolfram. Adjacency Matrix. Disponível em: <http://mathworld.wolfram.com/AdjacencyMatrix.html>.

MELO, Felipe Francisco Rios de. LEIJOTO, Thales Mrad. Coloração de grafos, 2019. Disponível em: <https://github.com/felipefrm/Coloracao-de-Grafos>.

JIA, Haixia. MOORE, Cristopher. How much backtracking does it take to color random graphs? Rigorous results on heavy tails, 2004. Disponível em: <https://www.cs.unm.edu/~hjia/heavytail.pdf>.