

Aplicação do Algoritmo de Coloração para a Separação de Produtos Químicos em Armazéns

Delphino Luciani de Paula Araújo Filho - 20219019597

Francisco Jhonnatas Ribeiro Santos - 20219022485

Lemuel Cavalcante Lopes - 20209063994

**Departamento de computação
Universidade Federal do Piauí - Teresina, PI - Brasil**

Abstract. *This article presents a possible application for the problem NP-Complete Graph Coloring, bringing an approach to separation of reagent chemicals, promoting greater safety for the factory or warehouse. in order to find heuristics and methods that help solve the problem*

Resumo. *Esse artigo traz uma possível aplicação para o problema NP-Completo Coloração de Grafos, trazendo uma abordagem para a separação de Produtos químicos reagentes, promovendo maior segurança para a fábrica ou armazém. a fim de encontrar heurísticas e métodos que ajudem a solucionar a problemática*

1.0 Introdução

O problema de coloração de grafos em armazenamento de produtos químicos envolve a atribuição de cores a tanques que representam vértices em um grafo. A proximidade entre tanques indica se eles podem conter produtos químicos compatíveis. O objetivo é evitar que tanques adjacentes tenham a mesma cor, minimizando o risco de reações químicas indesejadas e otimizando a segurança no armazenamento de substâncias químicas. Em resumo, trata-se de um problema de organização para garantir que produtos químicos incompatíveis estejam armazenados em tanques com cores diferentes.

O problema de Coloração de Grafos é um desafio fundamental na Teoria dos Grafos. Seu objetivo principal é atribuir cores aos vértices de um grafo de tal forma que vértices adjacentes não compartilhem a mesma cor. A tarefa envolve encontrar a menor quantidade de cores necessárias para colorir todo o grafo, o que significa buscar uma atribuição de cores que minimize o número total de cores utilizadas. Para ilustrar, considere a Figura 1 abaixo.

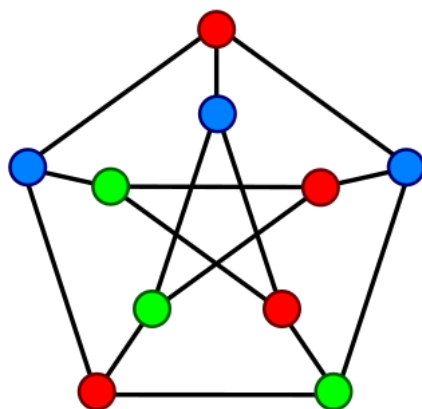


Figura 1. Exemplo de coloração de grafos(político).

1.1 Descrição do Problema

No contexto de armazenamento de produtos químicos em um armazém, a aplicação do algoritmo de coloração de grafos visa otimizar a organização e a segregação de produtos químicos com características diferentes. Neste cenário, cada produto químico é representado como um vértice no grafo, e as arestas do grafo denotam produtos químicos que não devem ser armazenados próximos uns dos outros devido a reações químicas indesejadas. Essa abordagem busca garantir a segurança, minimizar riscos e evitar interações indesejadas entre os produtos químicos.

Considere um exemplo em que um armazém precisa acomodar 8 produtos químicos distintos, cada um com suas características específicas. A lista de produtos químicos inclui:

1. Produto Químico A
2. Produto Químico B
3. Produto Químico C
4. Produto Químico D
5. Produto Químico E
6. Produto Químico F
7. Produto Químico G
8. Produto Químico H

Existem restrições de incompatibilidade entre alguns desses produtos químicos, o que significa que eles não podem ser armazenados na mesma área do armazém. Essas restrições são representadas por arestas no grafo. Por exemplo:

Produtos químicos que não devem ser armazenados próximos	
Produto Químico A	Não pode ser armazenado na mesma área que o Produto Químico C e o Produto Químico H.

Produto Químico B	Não pode ser armazenado na mesma área que o Produto Químico F e o Produto Químico G.
Produto Químico C	não pode ser armazenado na mesma área que o Produto Químico A e o Produto Químico H.
Produto Químico D	Não pode ser armazenado na mesma área que o Produto Químico E e o Produto Químico H.
Produto Químico E	não pode ser armazenado na mesma área que o Produto Químico F e o Produto Químico D.
Produto Químico F	Não pode ser armazenado na mesma área que o Produto Químico B e o Produto Químico E.
Produto Químico G	não pode ser armazenado na mesma área que o Produto Químico B e o Produto Químico H.
Produto Químico H	não pode ser armazenado na mesma área que o Produto Químico A, Produto Químico C, Produto Químico D e Produto Químico G.

Modelagem do Problema e Aplicação da Coloração de Grafos

Ao modelar esse problema e aplicar o algoritmo de coloração de grafos, obtemos uma representação visual da alocação dos produtos químicos nas áreas de armazenamento do armazém. Cada área do armazém é identificada por uma cor, e os produtos químicos são alocados às áreas de acordo com as restrições de incompatibilidade.

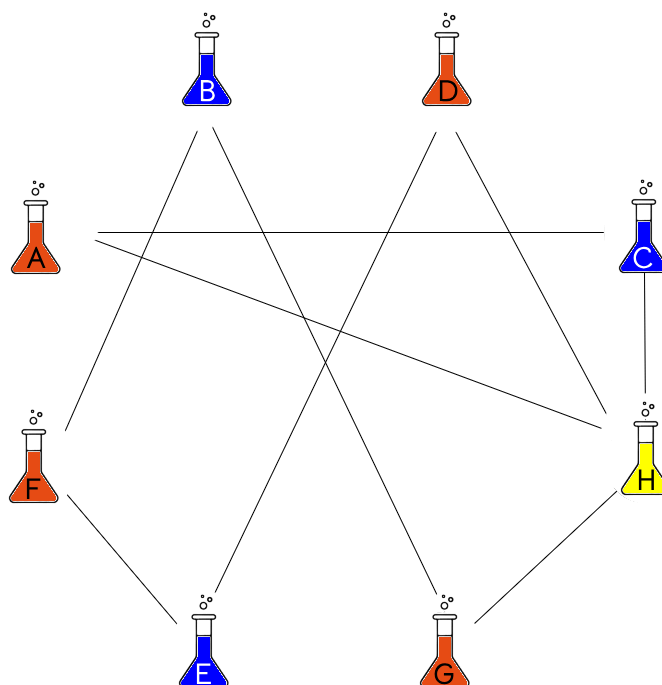


Figura 2. Exemplo de coloração de grafos(químico).


Após a análise das restrições e a aplicação da coloração de grafo, o resultado da alocação dos produtos químicos pode ser observado na representação visual do armazém. Os produtos químicos cujos vértices no grafo receberam uma cor específica foram designados

para a área de armazenamento correspondente, enquanto produtos químicos com cores diferentes são segregados em áreas separadas para garantir que não haja interações indesejadas entre eles.

Essa abordagem eficiente para a segregação de produtos químicos em armazéns promove a segurança e minimiza os riscos associados a reações químicas indesejadas, garantindo a organização e a eficácia no armazenamento de substâncias químicas diversas.

2.0 Implementação do algoritmo de Força Bruta

Para realizar a implementação do algoritmo de força bruta, foi essencial desenvolver diversas funções e variáveis que contribuíssem para a otimização do código. Isso incluiu a criação de uma matriz aleatória para formar a matriz de adjacência, bem como a capacidade de configurar manualmente o número de vértices na matriz. Com esses elementos preparatórios, a parte central do código, encarregada da coloração dos grafos, foi estruturada da seguinte maneira:

A screenshot of a code editor with a dark background and light-colored text. The code is a Python function named graphColoringAlgorithm. It includes global variables 'color' and 'colorArrayPrint'. The function uses recursion to try different color assignments for each vertex in the graph. It checks if a color assignment is safe (no conflicts with adjacent vertices) and if it leads to a complete coloring of the graph. The function returns True if a valid coloring is found, and False otherwise.

```
1 def graphColoringAlgorithm(graph, m, i, colorArray):
2     global color
3     global colorArrayPrint
4
5     if i == len(graph):
6         if isSafe(graph, colorArray):
7             colorArrayPrint = colorArray.copy()
8             color = len(set(colorArray))
9             return True
10        return False
11
12    for j in range(1, m + 1):
13        colorArray[i] = j
14        if graphColoringAlgorithm(graph, m, i + 1, colorArray):
15            return True
16        colorArray[i] = 0
17
18    return False
19
```

Figura 3. Função graphColoringAlgorithm.

2.1 Principais Características do Algoritmo

2.1.1 Função “ gerar_grafo(n) ”

Esta função recebe um número inteiro n como parâmetro e gera uma matriz de adjacência de tamanho $n \times n$ com valores aleatórios entre 0 e 1. Ela é usada para criar um grafo aleatório.

Ao finalizar sua execução, a função retorna a matriz de adjacência que sofrerá a coloração. Sua complexidade é: $O(n^2)$, pois ela percorre uma matriz quadrada de tamanho $n \times n$ e preenche os elementos com valores aleatórios. Portanto, o número de operações é proporcional a n^2 .

2.1.2 Função “ printConfiguration(colorArray) ”

Esta função recebe um array de cores Color Array como parâmetro e imprime na tela a cor atribuída a cada vértice do grafo. Sua complexidade é: $O(V)$, pois a função itera sobre os vértices do grafo (V) e imprime a configuração de cores para cada vértice. Portanto, a complexidade é linear em relação ao número de vértices.

2.1.3 Função “ isSafe(graph, colorArray) ”

Esta função recebe uma matriz de adjacência $graph$ e um array de cores $colorArray$ como parâmetros. Ela verifica se é seguro atribuir a cor de um vértice i a todos os seus vizinhos j , comparando a cor atribuída a i com a cor atribuída a j . Se a cor for a mesma, a função retorna False, indicando que não é seguro atribuir a cor a i . Se a cor for diferente para todos os vizinhos, a função retorna True, indicando que é seguro atribuir a cor a i . Sua complexidade é: $O(V^2)$, pois utiliza dois loops aninhados para percorrer todos os pares de vértices no grafo. Portanto, a complexidade é quadrática em relação ao número de vértices.

2.1.4 Função “ graphColoringAlgorithm(graph, m, i, colorArray) ”

Esta função tem como por objetivo implementar o algoritmo de coloração de grafos de forma recursiva. A função verifica se chegou ao último vértice e, se sim, verifica se a coloração atual é segura. Se for segura, armazena o vetor de cores, conta o número de cores distintas e retorna True. Caso contrário, retorna False. Se não for o último vértice, a função itera sobre as cores disponíveis e atribui uma cor ao vértice atual. Em seguida, chama recursivamente a função para o próximo vértice. Se a chamada recursiva retornar True, uma coloração válida foi encontrada, então retorna True. Caso contrário, redefine a cor do vértice atual e tenta a próxima cor. Se nenhuma cor for encontrada para o vértice atual, retorna False. Sua complexidade é: $O(m^V)$ visto que a complexidade dessa função depende do número de vértices “ V ” e do número de cores “ m ”, em seu pior caso todos os vértices são adjacentes e a função consequentemente percorre todas as possíveis combinações de cores.

2.1.5 Função “ main ”

Primeiro, gera um grafo aleatório e imprime informações sobre ele. Em seguida, inicializa uma lista de cores e itera várias vezes para tentar encontrar uma coloração válida. Para cada iteração, mede o tempo de início e chama a função de coloração do grafo. Se a coloração for bem-sucedida, imprime a configuração de cores, o número de cores utilizadas e o tempo de execução. Caso contrário, exibe uma mensagem informando que não foi possível colorir o grafo. Em resumo, o programa gera um grafo aleatório e tenta encontrar uma coloração válida para ele, exibindo o resultado e o tempo de execução. Sua complexidade é: $O(10)$ ou $O(1)$ pois essa função apenas depende do número de iterações no loop, que é definido como 10 no código.

2.1.6 Complexidade Geral do Código

A complexidade do código é definido pela função `graphColoringAlgorithm`, sendo então $O(mv)$ visto que as demais funções possuem complexidade menores ou irrelevantes. Em outras palavras, a função `graphColoringAlgorithm` é chamada recursivamente cada vértice “i” e para cada cor “j”, sendo que para cada vértice “i” há “m” cores para serem atribuídas. Como há V vértices, isso resulta em um número total de chamadas recursivas de mv sendo esta maior complexidade presente no algoritmo.

2.2 Testes

Foram conduzidos experimentos com o intuito de simular uma matriz de adjacência aleatória, visando a um estudo de caso mais aprofundado do problema em questão. Nesse contexto, foram realizados no mínimo 3 testes, nos quais foram empregadas matrizes contendo de 5 a 9 vértices.

2.2.1 Ferramentas utilizadas para a realização dos testes

Linguagem de Programação	Python
IDE	Vscode
Sistema Operacional	Windows 11
Processador	i3 1215-u , 1.2GHZ to 4.4GHZ, 6/8
Memória Ram	8GB 3200Mhz

2.2.1 Gráfico de execuções

Tempo(segundos)

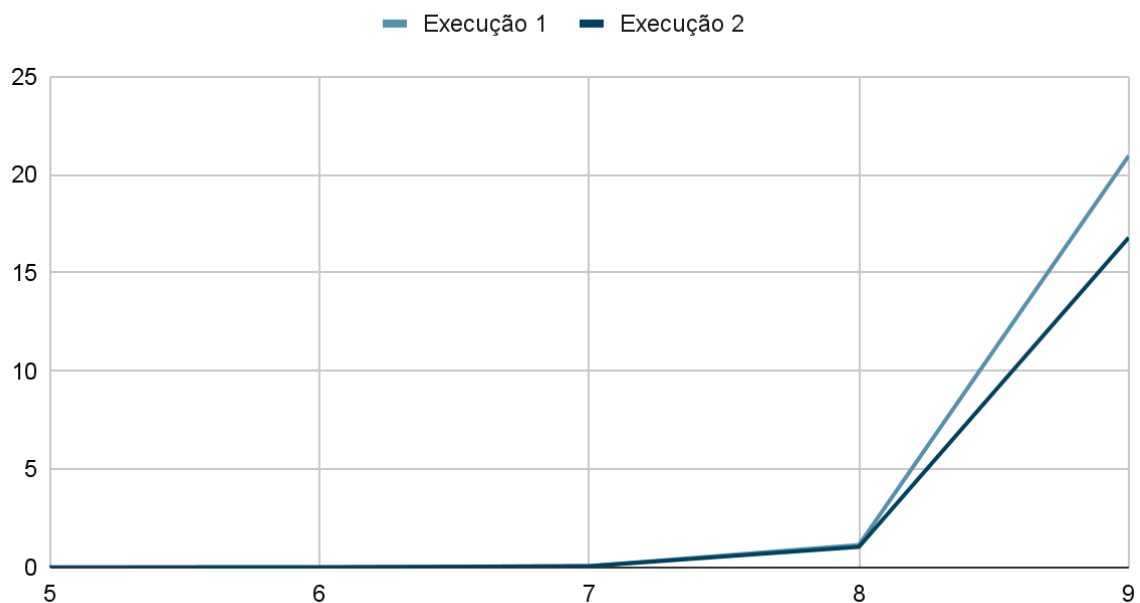


Figura 4. Tempo de execução utilizando força Bruta

Vértices	Execução 1	Execução 2	Execução 3
5	0,00203943	0,0093943	0,00103943
6	0,01010323	0,01210323	0,00910323
7	0,07202148	0,06702148	0,06902148
8	1,14801869	1,04801869	0,94801869
9	20,96455383	16,79729795	14,79729795

3. Problema NP-Completo - NP = nondeterministic polynomial time

São problemas computacionais intratáveis, para os quais não existem algoritmos que possam resolvê-los em tempo finito, mesmo quando dado um tempo de execução ilimitado. Um exemplo é o problema do Caixeiro Viajante, em que a intenção é passar por todos os vértices uma única vez e voltar ao vértice de origem.

O status dos problemas NP-Completo é caracterizado por uma incerteza semelhante. Problemas NP-completos são problemas para os quais não se sabe se existem algoritmos eficientes para resolvê-los. Formalmente, dado um problema de decisão L, ele é classificado como NP-completo se duas condições forem satisfeitas:

1. L pertence à classe NP, o que significa que soluções propostas para o problema podem ser verificadas em tempo polinomial, embora encontrar tais soluções não seja necessariamente eficiente.

2. Qualquer problema em NP pode ser transformado (por meio de uma redução) em L em tempo polinomial, indicando que L é pelo menos tão difícil quanto os outros problemas em NP.

Tomando o exemplo do problema "3-coloring", onde a tarefa é determinar se é possível atribuir três cores diferentes aos vértices de um grafo de modo que vértices adjacentes tenham cores diferentes, é possível demonstrar que o problema "K-coloring" é NP-completo utilizando as proposições.

3.1. Prova que é NP

A demonstração da NP-dificuldade do Problema 3-coloring envolve uma técnica de redução a partir de um problema já conhecido como NP-difícil, que é o 3-SAT. Suponhamos que tenhamos uma fórmula booleana no problema 3-SAT com m cláusulas envolvendo n variáveis denotadas por x_1, x_2, \dots, x_n .

Para criar o grafo associado ao Problema 3-coloring, seguimos os seguintes passos:

1. Cada variável x_i na fórmula resulta na criação de dois vértices no grafo: um vértice v_i para representar a variável x_i e outro vértice v_i' para representar a negação da variável x_i .
2. Para cada cláusula c em m , adicionamos cinco vértices ao grafo, denominados c_1, c_2, \dots, c_5 , representando os possíveis valores de verdade das cláusulas.
3. Adicionalmente, introduzimos três vértices coloridos com cores distintas: T (Verdadeiro), F (Falso) e N (Neutro), os quais denotam os valores de verdade possíveis.
4. Estabelecemos arestas entre os três vértices adicionais T, F e N, formando um triângulo.
5. Adicionamos arestas entre os vértices v_i e v_i' , assim como entre esses vértices e o vértice Neutro (N), criando triângulos induzidos que estão conectados.

Essa construção mapeia a estrutura da fórmula booleana 3-SAT para a configuração de vértices e arestas no grafo do Problema 3-coloring. Essa associação é projetada de forma que a verificação da coloração do grafo se relacione diretamente com a avaliação da fórmula booleana original. Essa técnica de redução permite demonstrar que, se conseguirmos resolver o Problema 3-coloring de maneira eficiente, também seríamos capazes de resolver o 3-SAT de maneira eficiente.

Dado que o 3-SAT é um problema NP-difícil, isso implica que o Problema 3-coloring também é NP-difícil.

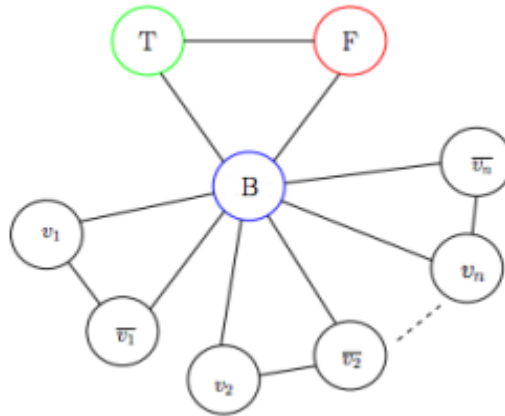


Figura 5. Grafo G gerado com a redução do problema 3-SAT ao 3-coloring

Se G for 3-colorível, então ou v_i ou v_i' recebe a cor T, e interpretamos isso como a atribuição de verdade para v_i . Agora, precisamos apenas adicionar restrições (arestas? vértices extras?) a G para capturar a satisfatibilidade das cláusulas da solução. Para fazer isso, é introduzido o Dispositivo de Satisfatibilidade de Cláusulas, também conhecido como OR-gadget. Essa construção é feita para representar graficamente a lógica booleana de maneira que a 3-coloração do grafo reflita a atribuição de verdade das variáveis e a satisfatibilidade das cláusulas na fórmula booleana original. O OR-gadget é um componente chave usado para modelar a operação lógica OR em termos de 3-coloração, garantindo que as cláusulas sejam satisfeitas se e somente se as variáveis apropriadas forem atribuídas como verdadeiras.

Com isso é provado que a instância 3sat é satisfeita e é provado que o problema pode ser resolvido utilizando o 3-coloring.

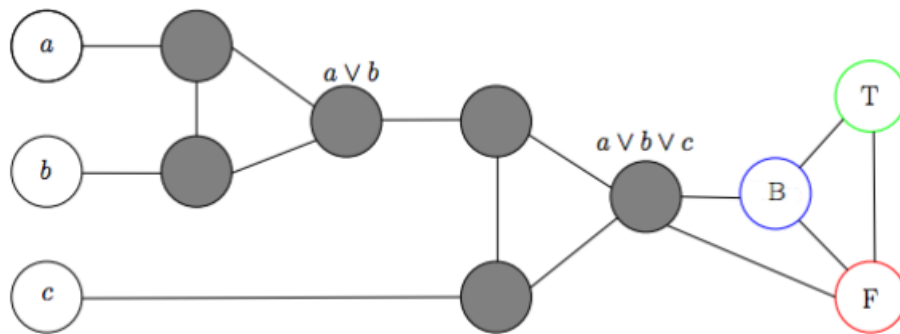


Figura 6. Problema 3sat reduzido ao 3-coloring

4. Implementação da Heurística

```
1 def heuristica_gulosa(grafo):
2     coloracao = {} # Dicionário para armazenar a cor de cada vértice
3
4     # Função para verificar se uma cor é segura para um vértice
5     def cor_segura(vertice, cor):
6         for vizinho in grafo[vertice]:
7             if vizinho in coloracao and coloracao[vizinho] == cor:
8                 return False
9         return True
10
11     # Itera sobre cada vértice do grafo
12     for vertice in grafo:
13         cores_usadas = set(coloracao[vizinho] for vizinho in grafo[vertice] if vizinho in coloracao)
14
15         # Testa cores a partir da cor 0
16         cor = 0
17         while cor in cores_usadas or not cor_segura(vertice, cor):
18             cor += 1
19
20         coloracao[vertice] = cor
21
22     return coloracao
```

Figura 7. Função principal do algoritmo heurístico

Nesta implementação, heurística gulosa recebe um grafo como entrada e retorna um dicionário que mapeia cada vértice para a cor atribuída a ele. A função cor segura verifica se uma cor é segura para um determinado vértice, ou seja, se não há conflitos de cor com os vértices adjacentes já coloridos.

Então, o algoritmo itera sobre cada vértice do grafo, atribuindo a menor cor possível que não cause conflitos de cor com os vértices adjacentes. Essa abordagem gulosa tenta encontrar uma solução aproximada eficiente para o problema de coloração de vértices.

Já a complexidade da heurística se dá por 3 loops presentes na função, sendo eles o loop externo no qual consiste nas chamadas da função para cada vértice, o loop interno em que consiste na iteração por todas as cores possíveis, indo de 1 até o número total de cores, m , sendo este igual a V , por fim temos a chamada da função `isSafe`, a mesma percorre todos os vértices no grafo (até ' V '), verificando se a atribuição da cor é segura, em outras palavras se não conflito com os vértices adjacentes, sendo então a complexidade de cada:

Complexidade do loop externo: $O(V)$

Complexidade do loop interno: $O(V)$

Complexidade da função `isSafe`: $O(V)$

Multiplicando essas 3 complexidades temos que a complexidade da função é $O(V^3)$.

5. Resultados

Para obter os resultados deste estudo, conduzimos uma série de testes automatizados, utilizando um gerador de matrizes de adjacência aleatórias e variando a quantidade de vértices para representar diferentes grupos de produtos químicos e suas interações.

A linguagem de programação escolhida para os testes foi Python, devido à sua praticidade, e os testes foram realizados no ambiente do Visual Studio Code. O sistema operacional utilizado foi o Windows 10 Pro, e o hardware consistiu em um processador Intel Core i3 de décima segunda geração, 8GB de memória RAM e um SSD de 256 GB.

Em cada iteração, foram realizados 5 testes com o mesmo número de vértices (V), a fim de calcular o tempo médio de execução. Todos os testes foram executados com entradas geradas aleatoriamente, utilizando uma função simples para criar matrizes de adjacência aleatórias.

5.1. Algoritmo de Força Bruta

Durante o período dos testes, ficou evidente que o algoritmo de força bruta não seria viável para entradas muito grandes, o que era esperado. Diante disso, optamos por realizar testes com entradas menores, aumentando gradualmente o tamanho dos grafos em intervalos modestos. O gráfico mostra a progressão do tempo de execução em relação ao tamanho da entrada. Observamos uma progressão exponencial do tempo a partir de 9 vértices. A partir de 11 vértices, os testes tornaram-se inviáveis devido ao tempo de execução extremamente longo.

5.2. Algoritmo com Heurística Gulosa

Ao realizarmos os primeiros testes, ficou clara a grande vantagem do algoritmo com heurística gulosa em relação ao de força bruta. Em geral o algoritmo de Heurística consegue computar no mesmo tempo de execução grafos 100 vezes maiores

Tempo(segundos)

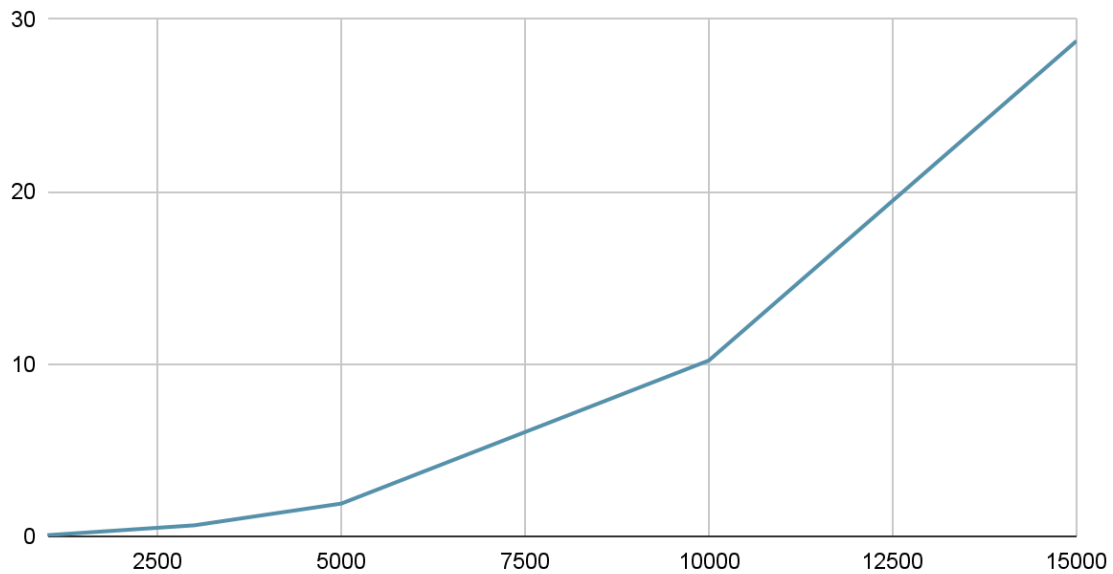


Figura 8. Gráfico tempo de execuções com método heurístico

Tempo(segundos)

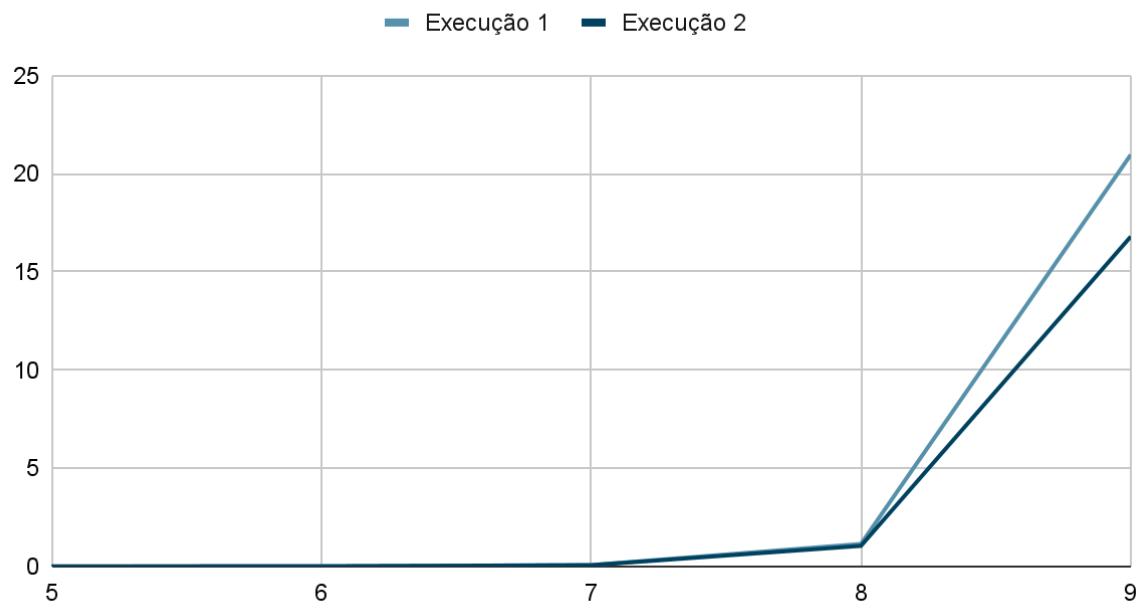


Figura 9. Tempo de execução utilizando força Bruta

5.3. Avaliação do Fator de Qualidade

Para comparar a eficácia relativa entre a abordagem heurística e a implementação de força bruta, empregamos o conceito do fator de qualidade, um parâmetro que nos permite mensurar a diferença de desempenho em termos de tempo de execução. O fator de qualidade,

cujo valor varia em uma escala contínua entre 0 e 1, reflete a magnitude da melhoria alcançada pela heurística em relação à abordagem de força bruta.

Quando o fator de qualidade é igual a 0, isso indica que os tempos de execução entre ambas as abordagens são praticamente equivalentes, ou seja, não houve ganho notável na eficiência da heurística em comparação com a abordagem de força bruta. Por outro lado, quando o fator de qualidade atinge o valor máximo de 1, isso denota que a heurística conseguiu otimizar o tempo de execução de maneira significativa, alcançando uma melhoria de 100% em relação à abordagem mais custosa, que é a força bruta. O cálculo do fator de qualidade é expresso pela fórmula:

$$\text{FatorDeQualidade} = ((\text{TempoForcaBruta} - \text{TempoHeuristica}) / \text{TempoForcaBruta})$$

Quantidade de vértices	Tempo gasto pela Força Bruta	Tempo Gasto pela Heurística	Fator de Qualidade
6	0,00910323	0.0009985 s	0.888
7	0,06902148	0.0009983 s	0.985
8	0,94801869	0.0009990 s	0.999
9	14,79729795	0.0009994 s	0.99993
10	152,80635902(2min 31 segundos)	0.0010002 s	0.999993

Figura 10. Fator de qualidade entre a implementação de força bruta e a implementação da heurística gulosa.

Nos experimentos conduzidos, que abrangeram casos com um número variável de vértices, entre 6 e 9 vértices, os resultados foram consistentes. Ao analisar os dados coletados, observamos que, com 6 vértices, o tempo gasto na coloração pela heurística foi o mais próximo do tempo gasto pela abordagem de força bruta. No entanto, a partir de 7 vértices, é evidente que o valor do fator de qualidade tende a 1. Isso sugere que a heurística demonstrou um desempenho superior em relação à implementação de força bruta, pelo menos dentro do conjunto de testes realizados.

Portanto, a análise do fator de qualidade, juntamente com os resultados consistentes obtidos nos experimentos, reforça a ideia de que a heurística é uma abordagem promissora para otimização de tempo de execução quando comparada à estratégia mais intensiva em recursos, representada pela abordagem de força bruta.

6. Conclusão

Com base nos dados coletados dos experimentos realizados com o algoritmo de força bruta e o algoritmo de heurística gulosa, concluímos que o algoritmo guloso apresenta resultados significativamente melhores para o problema de coloração de grafos. Mesmo considerando que a coloração de grafos é um problema NP-Completo, e embora o algoritmo heurístico não garanta sempre a solução ótima, determinamos que a abordagem gulosa é a mais adequada para nossa aplicação.

Em situações de urgência, onde a alocação rápida do algoritmo é crucial para garantir a segurança de empresas que lidam com produtos químicos, o algoritmo heurístico oferece uma solução eficaz. Portanto, concluímos que a abordagem gulosa é a escolha preferencial para resolver o problema de coloração de grafos no contexto de produtos químicos em armazéns.

7. Referências

Acervo Lima. Disponível em: <<https://acervolima.com/3-coloracao-e-np-complete/>> .
Acesso em: 30 Janeiro de 2024.

Algoritmos / Thomas H. Cormen... [et al.]; tradução Arlete Simille Marques]. - [Reimpr.]. - Rio de Janeiro: GEN | Grupo Editorial Nacional. Publicado pelo selo LTC | Livros Técnicos e Científicos Editora Ltda, 2022.