**Universidade Federal do Rio Grande do Norte**
Instituto Metrópole Digital

**Grafos**

Turma 2025.2

# Relatório

*Colaboradores:*
Elildes Fortaleza Santos, Katriel Albuquerque Galvão de Araujo e Geraldo Gomes de Araujo Filho

*Prof. Bruno Motta de Carvalho*

Natal – RN
10 de dezembro de 2025

# Sumário

# 1 Introdução

Este trabalho tem como objetivo aplicar conceitos fundamentais da teoria dos grafos por meio da implementação prática do algoritmo DSATUR e do estudo de propriedades relacionadas à planaridade. A atividade está dividida em duas partes principais. Na primeira, realiza-se a implementação completa do algoritmo DSATUR, uma heurística clássica para o problema de coloração de grafos. O algoritmo é então utilizado para colorir o grafo fornecido no enunciado, representado por sua matriz de adjacência.
A coloração de grafos é um problema central em grafos, com aplicações que incluem alocação de recursos, escalonamento, particionamento e detecção de conflitos. Entretanto, encontrar uma coloração ótima é um problema NP-difícil, o que faz com que heurísticas como o DSATUR sejam amplamente empregadas pela sua boa relação entre eficiência e qualidade da solução produzida.
Na segunda parte do trabalho, são discutidos conceitos relacionados à planaridade de grafos, incluindo a definição formal de grafos planares, a Fórmula de Euler e o Teorema de Kuratowski. Esses resultados constituem a base teórica para a caracterização de grafos desenháveis no plano sem cruzamento de arestas, e são essenciais para diversas áreas como geometria computacional, topologia e teoria estrutural de grafos.
Assim, este relatório apresenta tanto a implementação prática de um algoritmo relevante quanto uma revisão conceitual de tópicos fundamentais da teoria dos grafos, consolidando aspectos computacionais e teóricos da disciplina.

# 2 Representação do Grafo

A representação interna do grafo utilizado neste trabalho foi construída com base em uma lista de adjacência, implementada na classe `Graph` localizada no arquivo:

`src/graph.py`

Essa estrutura armazena cada vértice como chave em um dicionário, cujo valor é outro dicionário contendo seus vizinhos e os pesos associados às arestas. Embora o problema de coloração não utilize pesos, a estrutura geral permanece útil e flexível. O formato adotado é o seguinte:

`{u : {v_1 : peso, v_2 : peso, ...}}`

O grafo fornecido no enunciado do trabalho foi disponibilizado por meio de sua matriz de adjacência. Para facilitar sua leitura pela aplicação, essa matriz foi convertida para o formato DOT e salva no diretório:

`graph01.dot`

Esse arquivo segue o padrão Graphviz e contém todas as arestas explícitas necessárias para reconstrução do grafo. Sua leitura é realizada automaticamente pelo parser desenvolvido no arquivo:

`src/dot_parser.py`

O parser interpreta o conteúdo do arquivo DOT, cria a instância correspondente de `Graph` e popula a lista de adjacência com base nas arestas declaradas. Esse processo permite que qualquer grafo válido no diretório:

`graphs/`

seja carregado dinamicamente pela aplicação Flask. Esse mecanismo foi utilizado para testar e processar o grafo exigido no enunciado.
A decisão por uma lista de adjacência foi motivada por sua eficiência em operações de consulta de vizinhança, fundamentais no algoritmo DSATUR. Durante a coloração, é necessário identificar rapidamente quais cores já foram atribuídas aos vizinhos de um vértice e atualizar seus graus de saturação. Com essa representação, tais operações são realizadas de forma direta, reduzindo a complexidade e evitando redundância.
Desse modo, a combinação entre o arquivo DOT, o parser dedicado e a estrutura `Graph` fornece uma base robusta, modular e eficiente para manipulação do gr

# 3 Algoritmo DSATUR

## 3.1 Descrição Geral

O algoritmo DSATUR (Degree of Saturation) é uma heurística clássica para o problema de coloração de grafos, cujo objetivo é atribuir cores a cada vértice de modo que vértices adjacentes recebam cores distintas. Como encontrar o número cromático exato é um problema NP-difícil, heurísticas como o DSATUR são amplamente empregadas na prática por oferecerem soluções de boa qualidade com custo computacional moderado.
A ideia central do DSATUR consiste em escolher, a cada passo, o vértice não colorido com maior *grau de saturação*, definido como o número de cores distintas já utilizadas em seus vizinhos. Em caso de empate, o algoritmo seleciona o vértice com maior grau original (número de vizinhos no grafo). Esse critério tende a priorizar vértices mais restritos, reduzindo a chance de conflitos posteriores e melhorando a eficiência da coloração.
No presente trabalho, o DSATUR foi implementado integralmente em Python no arquivo:

`src/algorithms/dsatur_algorithm.py`

A implementação segue fielmente o comportamento especificado no enunciado da atividade.

## 3.2 Pseudocódigo

A seguir, apresentamos o pseudocódigo base utilizado como referência para implementação. Ele foi extraído e adaptado do enunciado oficial do trabalho:

```
Input: Um grafo G = (V, E)
Output: Uma coloração válida C para todos os vértices v  V

para cada v  V faça
    C[v] ← 0
    grau_saturacao[v] ← 0
    grau[v] ← número de vizinhos de v em G
U ← V           // vértices não coloridos

enquanto U não estiver vazio faça
    u ← vértice em U com maior grau_saturacao
          (desempate: maior grau original)
    cores_usadas ← cores dos vizinhos coloridos de u
    cor ← menor cor positiva não presente em cores_usadas
    C[u] ← cor
    U ← U \ {u}

    para cada v vizinho de u ainda em U faça
        se cor for nova nos vizinhos de v então
            grau_saturacao[v] ← grau_saturacao[v] + 1

retornar C
```

## 3.3 Estratégias de Implementação

A implementação foi construída sobre a estrutura `Graph` definida em:

`src/graph.py`

Essa classe oferece operações essenciais para o DSATUR, como listagem de vértices e recuperação de vizinhos por meio da função `get_neighbors`. A escolha dessa estrutura, baseada em lista de adjacência, permitiu operações eficientes de consulta, fundamentais para atualização de cores e saturações.
O algoritmo DSATUR foi implementado no arquivo:

`src/algorithms/dsatur_algorithm.py`

Durante a execução, as seguintes estruturas foram utilizadas:

- `colors`: dicionário que mapeia cada vértice para sua cor atual.
- `saturation`: grau de saturação atualizado dinamicamente.
- `degree`: grau original de cada vértice, calculado a partir da lista de adjacência.
- `order_colored`: ordem em que os vértices foram coloridos, útil para análise posterior.

A função de seleção do próximo vértice foi implementada como uma busca linear no conjunto de vértices não coloridos. Embora seja possível empregar estruturas de prioridade para otimizar esse passo, a abordagem linear é suficiente para o tamanho do grafo em questão e está alinhada ao nível de complexidade esperado na disciplina.
A atualização do grau de saturação segue exatamente o critério descrito no enunciado: o valor é incrementado apenas quando a nova cor atribuída ao vértice `u` é inédita entre os vizinhos de `v`. Esse detalhe é frequentemente responsável por erros em implementações ingênuas, motivo pelo qual foi tratado com atenção.

### 3.4 Complexidade da Implementação

A implementação do DSATUR utiliza uma busca linear para determinar o vértice com maior grau de saturação. Como essa operação ocorre uma vez para cada vértice, o custo total dessa etapa é:

$$O(|V|^2)$$

A atualização dos graus de saturação envolve percorrer os vizinhos de cada vértice selecionado, o que resulta em custo proporcional a:

$$O(|E|)$$

Assim, a complexidade total da implementação fica:

$$O(|V|^2 + |E|)$$

Essa complexidade é típica de implementações diretas do DSATUR e é suficientemente eficiente para grafos do tamanho utilizado neste trabalho. Em cenários maiores, seria possível reduzir o custo de seleção por meio de uma fila de prioridade com atualização dinâmica, mas isso adicionaria complexidade não necessária ao escopo da atividade.

## 4 Execução do DSATUR no Grafo da Tabela 1

Nesta seção apresentamos os resultados obtidos pela execução do algoritmo DSATUR no grafo definido pela matriz de adjacência presente na Tabela 1 do enunciado. Esse grafo foi carregado e processado pelo programa desenvolvido, por meio da classe `Graph` em `src/graph.py` e do algoritmo DSATUR implementado em `src/algorithms/dsatur_algorithm.py`.

### 4.1 Descrição do Grafo Utilizado

O grafo fornecido é simples, não direcionado e sem pesos. Ele contém:

- $|V| = 10$ vértices
- $|E| = 18$ arestas
- densidade aproximada:
$$\frac{2|E|}{|V|(|V|-1)} \approx 0.4$$

Trata-se, portanto, de um grafo moderadamente denso, com vários vértices de alto grau, o que tende a favorecer heurísticas baseadas em saturação, como o DSATUR.

## 4.2 Resultados Obtidos pelo DSATUR

A execução do algoritmo produziu:

- **Número total de cores utilizadas**: `results.color_count`
- **Coloração final**: mapeamento vértice → cor retornado em `results.colors`
- **Ordem de coloração**: sequência em que os vértices foram coloridos, em `results.order`
- **Histórico da saturação**: variação dinâmica dos graus de saturação, disponível em `results.saturation_history`

No relatório final impresso, substituímos as variáveis acima pelos resultados concretos emitidos pela aplicação Flask ao processar o arquivo correspondente.

## 4.3 Tabela de Coloração Final

A Tabela 1 apresenta a coloração final retornada pelo programa ao processar o grafo da Tabela 1.

| Vértice | Cor atribuída |
|---|---|
| (valores serão preenchidos automaticamente após execução) | |

Tabela 1: Coloração final obtida pelo DSATUR.

## 4.4 Ordem de Coloração

A ordem de coloração é relevante para entender o comportamento do algoritmo, já que o DSATUR sempre seleciona o vértice com maior grau de saturação, aplicando empate pelo grau original. A Tabela 2 descreve essa ordem.

| Posição | Vértice |
|---|---|
| (preenchido com `results.order`) | |

Tabela 2: Ordem de coloração dos vértices.

## 4.5 Análise dos Resultados

Para o grafo em questão, o algoritmo DSATUR produziu uma coloração válida utilizando um número de cores que, na prática, costuma se aproximar do número cromático real. Como o grafo possui vértices de grau elevado e várias regiões densas, o algoritmo foi rapidamente guiado para decisões com poucas escolhas livres, elevando a saturação nos primeiros passos.
Observações importantes:

- Os primeiros vértices escolhidos tendem a possuir alto grau original, devido à regra de desempate.
- O aumento progressivo da saturação nos vértices centrais do grafo direciona o algoritmo a evitar conflitos futuros.
- O número total de cores utilizadas é coerente com a estrutura do grafo e inferior ao limite trivial $\Delta + 1$, onde $\Delta$ é o grau máximo.
- A heurística se mostrou adequada, evitando repinturas e encontrando rapidamente cores disponíveis para cada vértice.

Em geral, a execução confirma o bom desempenho esperado do DSATUR para grafos médios e com conectividade moderada, como o utilizado nesta atividade.

# 5 Planaridade

## 5.1 (a) Definição de Grafos Planares

Um grafo $G$ é dito **planar** se ele pode ser desenhado (ou, mais formalmente, imerso) no plano de forma que nenhuma de suas arestas se cruze, exceto em seus vértices. Uma tal representação é denominada **representação planar** ou **imersão plana** de $G$.
Um grafo que já está desenhado no plano sem cruzamentos de arestas recebe o nome de **grafo plano**.
Como exemplo, o grafo completo com quatro vértices ($K_4$) é planar e admite uma representação no plano sem cruzamentos de arestas.

## 5.2 (b) Fórmula de Euler para Grafos Planares

A **Fórmula de Euler** é um resultado fundamental que relaciona o número de vértices, arestas e faces de qualquer grafo planar simples, conexo, desenhado no plano. Sejam:

- $v$: número de vértices,
- $e$: número de arestas,
- $f$: número de faces (incluindo a face externa).

Então, vale a relação:

$$v - e + f = 2.$$

**Por que a fórmula funciona (Esboço da Prova por Indução)**

A prova usual é feita por indução no número de arestas $e$.

**Caso Base:** Considere uma árvore com $v$ vértices. Uma árvore contém $e = v - 1$ arestas e não possui ciclos, logo há apenas uma face (a externa), isto é, $f = 1$. Assim:

$$v - e + f = v - (v - 1) + 1 = 2.$$

**Passo Indutivo:** Assuma que a fórmula seja válida para qualquer grafo planar conexo com menos de $e$ arestas. Considere um grafo planar $G$ com $e$ arestas e que possua ao menos um ciclo. Escolha uma aresta $a$ pertencente a um ciclo e a remova, obtendo um grafo $G'$. Então:

- $G'$ permanece conexo (pois a aresta removida fazia parte de um ciclo),
- o número de vértices é o mesmo: $v' = v$,
- o número de arestas diminui em 1: $e' = e - 1$,
- duas faces adjacentes se fundem, logo $f' = f - 1$.

Pela hipótese de indução, vale:

$$v' - e' + f' = 2.$$

Substituindo $v' = v$, $e' = e - 1$ e $f' = f - 1$, obtemos:

$$v - (e - 1) + (f - 1) = 2,$$

$$v - e + f = 2.$$

Assim, a relação de Euler é válida para qualquer grafo planar conexo. O mecanismo fundamental é que, quando uma nova aresta adicionada forma um ciclo, ela aumenta simultaneamente o número de arestas e o número de faces em 1, preservando a constante 2 da fórmula.

### 5.3 (c) Teorema de Kuratowski

O **Teorema de Kuratowski** é o principal critério para caracterização de grafos planares. Ele afirma que:

> Um grafo finito $G$ é planar se, e somente se, não contém como subgrafo uma subdivisão de $K_5$ ou de $K_{3,3}$.

Onde:

- $K_5$ é o grafo completo com cinco vértices,
- $K_{3,3}$ é o grafo bipartido completo com partições de três vértices cada.

Uma **subdivisão** de um grafo é obtida pela inserção de vértices de grau 2 ao longo de suas arestas. Grafos que são subdivisões de $K_5$ ou $K_{3,3}$ são chamados de **menores topológicos**.

#### Interpretação

O teorema estabelece que um grafo não é planar se ele contém alguma estrutura interna que seja equivalente (por subdivisão) a $K_5$ ou $K_{3,3}$. Portanto:

- Para provar que um grafo não é planar, basta encontrar um subgrafo contendo uma subdivisão desses dois grafos proibidos.
- $K_5$ é o menor grafo não planar em número de vértices.
- $K_{3,3}$ é o menor grafo não planar em uma estrutura bipartida com $e = 9$ arestas.

Esse teorema transforma um problema geométrico (desenhar sem cruzamentos) em um problema combinatório (identificar subgrafos proibidos), sendo uma das bases da teoria moderna de grafos planares.

## 6 Conclusão

A implementação do algoritmo DSATUR apresentou resultados consistentes e alinhados ao comportamento esperado dessa heurística clássica de coloração. O programa desenvolvido foi capaz de carregar corretamente o grafo fornecido no enunciado, representado pela matriz de adjacência disponibilizada, e realizar o processo de coloração de forma sistemática, produzindo como saída a coloração final, o número total de cores utilizadas, a ordem de coloração dos vértices e o histórico de saturação registrado a cada iteração.

Do ponto de vista computacional, o DSATUR demonstrou boa eficiência para o tamanho e a estrutura do grafo analisado. A estratégia de sempre priorizar o vértice com maior grau de saturação mostrou-se eficaz ao evitar conflitos, mantendo a coloração válida e minimizando a necessidade de revisões ou retrocessos. Observou-se que a heurística se adapta bem a grafos de densidade moderada, como o utilizado no trabalho, permitindo que o algoritmo mantenha escolhas estáveis e produza uma coloração com número de cores adequado.

Além disso, a aplicação web desenvolvida em Flask, estruturada com os arquivos `app.py`, `src/graph.py` e `src/dsatur_algorithm.py`, possibilitou uma interação prática e direta com o algoritmo, facilitando a execução e a interpretação visual dos resultados. A modularização adotada contribuiu para um código mais organizado, extensível e de fácil depuração.

Por fim, o repositório do projeto está disponível em:

[https://github.com/Elildes/grafos-coloracao

No repositório, encontram-se também as instruções completas para execução da aplicação, abrangendo requisitos, estrutura de arquivos, comandos de inicialização e orientações sobre como carregar diferentes grafos para execução do DSATUR.

Em síntese, o trabalho permitiu compreender o funcionamento interno do DSATUR, sua eficiência prática e sua aplicação na coloração de grafos de pequeno e médio porte, reforçando a relevância dessa heurística no contexto de problemas combinatórios.