UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
NATAL/RN


# PROJETO PRÁTICO -
# COLORAÇÃO EM GRAFOS COM ALGORITMO *DSATUR*

Alunos:
CARLOS ALBERTO DE LIMA NETO
LUIZ GUILHERME CARVALHO VIANA
LUTÉRCIO JACKSON GUIMARÃES FILHO

Professor: BRUNO MOTTA DE CARVALHO

# 1. INTRODUÇÃO

A Teoria dos Grafos é uma área da matemática que desempenha um papel extremamente importante na modelagem e solução de problemas em diversas áreas, desde a otimização de redes e comunicação até a arquitetura de hardware. Entre os vários desafios que podem ser mapeados para a estrutura de grafos, o problema da Coloração de Vértices destaca-se por sua relevância prática e sua complexidade teórica. A coloração de um grafo consiste na atribuição de cores aos seus vértices, garantindo que nenhum par de vértices adjacentes (conectados por uma aresta) receba a mesma cor. O objetivo é utilizar o menor número possível de cores, um valor conhecido como *Número Cromático*. Por ser um problema classificado como NP-difícil, encontrar a solução ótima exata é impraticável para a maioria dos grafos grandes.

Dada a intratabilidade do problema ótimo, este trabalho se concentra na aplicação de um algoritmo guloso para obter uma solução de coloração próxima da máxima eficiência. O algoritmo escolhido é o DSATUR (*Degree of Saturation*). Este método prioriza a coloração dos vértices que impõem maior restrição aos seus vizinhos, medidos pelo seu grau de saturação (o número de cores distintas já utilizadas nos vértices vizinhos). O presente relatório descreve a implementação do algoritmo DSATUR e detalha sua utilização no cálculo da coloração de um grafo, analisando as decisões e a complexidade computacional da implementação para o problema proposto.

# 2. ANÁLISE TEÓRICA

## 2.1 Teoria dos Grafos: Conceitos Básicos

A Teoria dos Grafos fornece uma estrutura matemática poderosa que permite o estudo das relações, fornecendo uma ferramenta analítica que pode ser utilizada para a modelagem de sistemas complexos e relações de paridade entre objetos. Um grafo *G* é formalmente definido como um par ordenado *G = (V, E)*, onde *V* representa um conjunto finito e não vazio de vértices e *E* representa um conjunto de arestas, que são pares de vértices, comumente definidas por {u, v}, com *u, v pertencente à V,* que estabelecem a conexão ou relação entre esses nós. Duas arestas são ditas adjacentes se compartilham um vértice comum, e dois vértices são ditos adjacentes se estão conectados por uma aresta.

A conectividade local de um vértice é quantificada pelo seu *grau*, que corresponde ao número de arestas incidentes a ele. Essa métrica é crucial, pois vértices com alto grau indicam pontos de alta interação ou restrição no sistema, e essa medida é fundamental para heurísticas que buscam priorizar vértices de alta conectividade.

## 2.2 A problemática da Coloração de Vértices

O problema da Coloração de Vértices, por sua vez, exige a atribuição de um rótulo (cor) a cada vértice *v pertencente à V* de um grafo *G* de modo que a propriedade fundamental da problemática seja sempre satisfeita: vértices adjacentes nunca podem ter a mesma cor. O desafio gira em torno da tentativa de encontrar o menor número de cores que satisfaz essa condição, um valor que recebe o nome de *Número Cromático,* ou $\chi(G)$. A determinação do $\chi(G)$ é o objetivo central da coloração, pois ele representa o limite de otimização possível para um problema modelado. Contudo, a dificuldade da solução está na natureza combinatória do problema, que indica um crescimento exponencial do número de possíveis colorações à partir do aumento do número de vértices, tornando a busca exaustiva inviável.

A complexidade teórica do problema é dada por sua classificação como NP-difícil, categoria da Teoria da Complexidade que engloba os problemas cuja solução pode ser

verificada em tempo polinomial, mesmo que a solução completa possa levar um tempo exponencial. A impossibilidade de encontrar um algoritmo de tempo polinomial para determinar $x(G)$ impulsiona a utilização de métodos de aproximação, ou seja, em vez de buscar a solução exata, o objetivo prático se desloca para a obtenção de uma solução aproximada $x'(G)$ em um tempo computacional viável, para todas as instâncias do grafo. Geralmente, a solução de problemas do tipo é através de algoritmos heurísticos ou gulosos (*greedy*), como é o caso do DSATUR abordado neste relatório.

## 2.3 O algoritmo DSATUR

O nome DSATUR caracteriza um algoritmo guloso cujo princípio é priorizar a coloração dos vértices que impõem maior restrição. O algoritmo opera iterativamente, selecionando a cada passo o vértice não colorido que possui o maior Grau de Saturação (DSAT), que é o número de cores distintas já utilizadas por seus vizinhos, ou o vértice mais pressionado. Essa métrica garante que os pontos críticos do grafo sejam tratados primeiro. Uma vez escolhido, o vértice recebe a menor cor disponível que não esteja em conflito com seus vizinhos, resultando em uma coloração válida, que é frequentemente uma boa aproximação do *Número Cromático* $x(G)$.

# 3. ANÁLISE PRÁTICA

## 3.1 Implementação e Complexidade

O armazenamento do grafo solicitado na atividade durante a implementação do código foi realizado por meio de um

```
std::map<std::string, std::set<std::string>>
```

que guardou as informações do arquivo.dot e originou uma lista de adjacência $L$, lista essa que é passada como argumento para o funcionamento do DSATUR. Em seguida, essa lista de adjacência é convertida para uma representação com índices numéricos (índices inteiros de 0 a n−1), que é a forma usada internamente pelo DSATUR para manter vetores de cor, saturação, grau e a própria lista de vizinhos, tornando as operações do algoritmo mais simples e eficientes.

- Estruturas do algoritmo

```
// Estruturas DSATUR
    std::vector<std::vector<int>> adj(n);
    std::vector<int> color(n, -1), saturation(n, 0), degree(n, 0);
    std::vector<bool> colored(n, false);

    // Constrói adj numérica + graus
    for (const auto& [u_str, viz] : adj_map) {
        int u = nome_id.at(u_str);
        degree[u] = viz.size();
        for (const std::string& v_str : viz) {
            int v = nome_id.at(v_str);
            adj[u].push_back(v);
        }
    }
```

O algoritmo seguiu um padrão de implementação com algumas estruturas bem definidas, listadas a seguir:

- *adj*
  Lista de adjacência numérica;
  Cada posição *adj[u]* guarda um *vector<int>* com os índices dos vizinhos do vértice *u*.

- *color*
  Vetor de cores dos vértices.
  *color[u]* = *-1* significa que o vértice *u* ainda não foi colorido.
  Quando o algoritmo atribui uma cor, *color[u]* passa a ser um inteiro 0, 1, 2, ....
- *saturation*
  Grau de saturação de cada vértice.
  *saturation[u]* é a quantidade de cores diferentes presentes nos vizinhos já coloridos de *u*.
  Esse valor é usado como principal critério para escolher o próximo vértice a ser colorido no DSATUR.

- *degree*
  Grau do vértice.
  *degree[u]* é preenchido a partir do tamanho da lista de vizinhos de *u* e é usado como critério de desempate quando dois vértices têm a mesma saturação.

- *colored*
  Marca se o vértice já foi definitivamente colorido.
  *colored[u]* = *false* no início; depois que o algoritmo escolhe uma cor para *u*, passa para true e esse vértice deixa de ser considerado nas próximas escolhas.

- O que armazena a Priority Queue?

```
// Priority queue: (saturação, grau, -id) max-heap
  auto comp = [](const auto& a, const auto& b) {
     return std::get<0>(a) < std::get<0>(b) ||
           (std::get<0>(a) == std::get<0>(b) &&
            (std::get<1>(a) < std::get<1>(b) ||
             (std::get<1>(a) == std::get<1>(b) && std::get<2>(a) > std::get<2>(b))));
  };
  std::priority_queue<std::tuple<int,int,int>,
                      std::vector<std::tuple<int,int,int>>, decltype(comp)> pq(comp);
```

A fila de prioridade guarda tuplas *(saturação, grau, -id)* para cada vértice. A lambda *comp* está definida de forma que a priority queue vire um *max-heap* em relação a esses critérios, na ordem:

1. Maior saturação primeiro *(std::get<0>)*.
2. Em empate de saturação, maior grau *(std::get<1>)*.
3. Em empate de saturação e grau, maior id (como está usando *-id*, o maior ID real vira o menor *-id*, e a comparação *std::get<2>(a) > std::get<2>(b)* faz esse objeto ter prioridade).

Em C++, o *std::priority_queue* por padrão usa o comparador com uma semântica semelhante à “a vem depois de b?”, de tal maneira que *comp* foi construído para que o “maior” em saturação, depois grau, depois ID, fique no topo.

- Inicialização da Priority Queue

```
// Inicializa fila (prioriza nó inicial)
    for (int i = 0; i < n; ++i) {
        pq.push({0, degree[i], -i});
    }
```

Para a inicialização da fila, utilizamos uma estrutura que insere todos os vértices na fila com saturação inicial 0. A prioridade inicial segue uma lógica que valoriza os vértices de maior grau, com desempate por maior ID via `-i`, assim contribuindo para um resultado que faz a coloração acontecer, primeiramente, aos vértices de maior grau.

- Loop Principal de Coloração

```
while (!pq.empty()) {
        auto [_, __, neg_u] = pq.top(); pq.pop();
        int u = -neg_u;
        if (colored[u]) continue;

        // Menor cor válida
        std::set<int> used;
        for (int v : adj[u]) {
            if (color[v] != -1) used.insert(color[v]);
        }
        int cor = 0;
        while (used.count(cor)) ++cor;
        color[u] = cor;
        colored[u] = true;
```

Durante a coloração, utilizamos um *for* para processar os vértices por ordem de prioridade, até que todos estejam coloridos. Seguimos a lógica descrita abaixo:

**Passos:**

1. Extrai topo: *pq.top()* pega a maior prioridade, *pop()* remove;
2. Recupera ID: *u* = *-neg_u* (inverte o *-i* armazenado);
3. Ignora duplicatas: *if (colored[u]) continue;* para pular vértices já processados;
4. Escolhe cor: Menor cor não usada pelos vizinhos já coloridos;
5. Marca como feito: *color[u] = cor; colored[u] = true;*

**Resultado:** *color[]* contém coloração válida do grafo.

- Atualização dos vizinhos

```
// Atualiza saturação dos vizinhos
        for (int v : adj[u]) {
            if (!colored[v]) {
                saturation[v]++;
                pq.push({saturation[v], degree[v], -v});
            }
        }
    }
```

Após colorir *u*, o algoritmo atualiza todos os vizinhos não-coloridos na próxima iteração. Seguimos a lógica descrita abaixo:

**Passos:**

1. Para cada vizinho v de *u*: Percorre lista de adjacência
2. Se v não colorido ainda: *!colored[v]* filtra só pendentes
3. *saturation[v]++*: +1 cor distinta na vizinhança de v
4. Reinsere na fila: *pq.push()* com nova saturação (pode duplicar entradas)

**Resultado:** Vizinhos de *u* ganham maior prioridade na próxima escolha.

- Complexidade do Algoritmo

A eficiência computacional da nossa implementação do algoritmo DSATUR é determinada pelo uso de uma Priority Queue (Max-Heap) para gerenciar o Grau de Saturação (DSAT) dos vértices não coloridos. Esta otimização permite que a complexidade de tempo seja reduzida significativamente em comparação com a implementação padrão O(|V|^2).

| Tipo de Grafo | E ≈ | Complexidade Total |
|---|---|---|
| Disperso | V | O(V log V) |
| Denso | $V^2$ | O($V^2$ log V) |

Logo, a complexidade de tempo total é **O((V+E) log(V+E))**, porque temos V pushes iniciais + 2E pushes no loop de vizinhos × log Q do heap binário, sendo Q = (V + E).

## 3.2 Análise dos Resultados

Após a execução do algoritmo DSATUR sobre o grafo definido na atividade, o número total de cores necessárias para obter uma coloração válida foi $x(G)$ = X. O *número cromático* atingido representa a solução aproximada encontrada pela heurística gulosa, evidenciando o possível limite de otimização para o problema.

Para compilar e usar o código, utilizar o seguinte comando:

```
g++ dsatur.cpp -lboost_graph -lboost_regex -o dsatur && ./dsatur
```

# 4. CONCLUSÃO

O presente trabalho demonstrou a implementação e aplicação do algoritmo DSATUR, uma heurística gulosa essencial para a obtenção de soluções viáveis no contexto do problema de Coloração de Vértices. Dada a complexidade teórica do problema, que é classificado como NP-difícil, a utilização de métodos de aproximação como o DSATUR prova ser uma estratégia eficaz para contornar a intratabilidade da determinação exata do *Número Cromático* $x(G)$.

A implementação do algoritmo no grafo fornecido resultou em uma coloração válida utilizando **x** cores. A análise da complexidade de tempo revelou que a solução implementada opera de maneira mais simplificada do que a implementação original, garantindo que o algoritmo seja computacionalmente eficiente para grafos de tamanho moderado. As escolhas do algoritmo, como a estrutura de dados para representação do grafo e a priorização do grau de saturação, foram fundamentais para a performance exemplar da heurística.

# 5. REFERÊNCIAS


DSatur Algorithm for Graph Coloring - GeeksforGeeks

DSatur - Wikipedia