COLORAÇÃO DE GRAFOS

COMPLEXIDADE DE ALGORITMOS

Estudante: Nicolle Beatrice Asquino

1. OBJETIVO

- Implementar dois algoritmos para colorir o grafo:
- Backtracking: encontra a solução ótima (menor número de cores).
- Algoritmo Guloso: rápido, aproximação que pode usar mais cores.

2. METODOLOGIA

- O grafo foi fornecido no moodle como matriz de adjacência em arquivo CSV.
- Backtracking: tenta atribuir cores recursivamente e faz "backtrack" quando não é possível.
- Guloso: percorre os vértices sequencialmente e atribui a menor cor disponível.
- As cores atribuídas foram exportadas para um arquivo texto e visualizadas em Python.

3. ALGORITMOS

Algoritmo Backtracking

isSafe

- Verifica se é seguro atribuir a cor c ao vértice v.
- Percorre todos os vértices e retorna false se algum vértice adjacente já tiver a mesma cor.
- Retorna true se não houver conflito.

graphColoringBT

- Implementa o algoritmo de backtracking para coloração de grafos.
- Tenta colorir o vértice v com cada cor de 1 até m.
- Se isSafe retornar true, atribui a cor e chama recursivamente para o próximo vértice (v + 1).
- Se não houver cor possível, volta atrás atribuindo 0 e tentando outra cor.
- Retorna true se todos os vértices foram coloridos corretamente; caso contrário, retorna false.

```c
int isSafe(int v, int c) {
    for (int i = 0; i < N; i++) {
        if (graph[v][i] && color[i] == c) {
            return 0;
        }
    }
    return 1;
}

int graphColoringBT(int v, int m) {
    if (v == N) return 1;

    for (int i = 1; i <= m; i++) {
        if (isSafe(v, i)) {
            color[v] = i;
            if (graphColoringBT(v + 1, m)) return 1;
            color[v] = 0;
        }
    }
    return 0;
}
```

Algoritmo Guloso

- Todos os vértices começam sem cor (-1).
- O primeiro vértice (0) recebe a primeira cor (0).

Processo para cada vértice

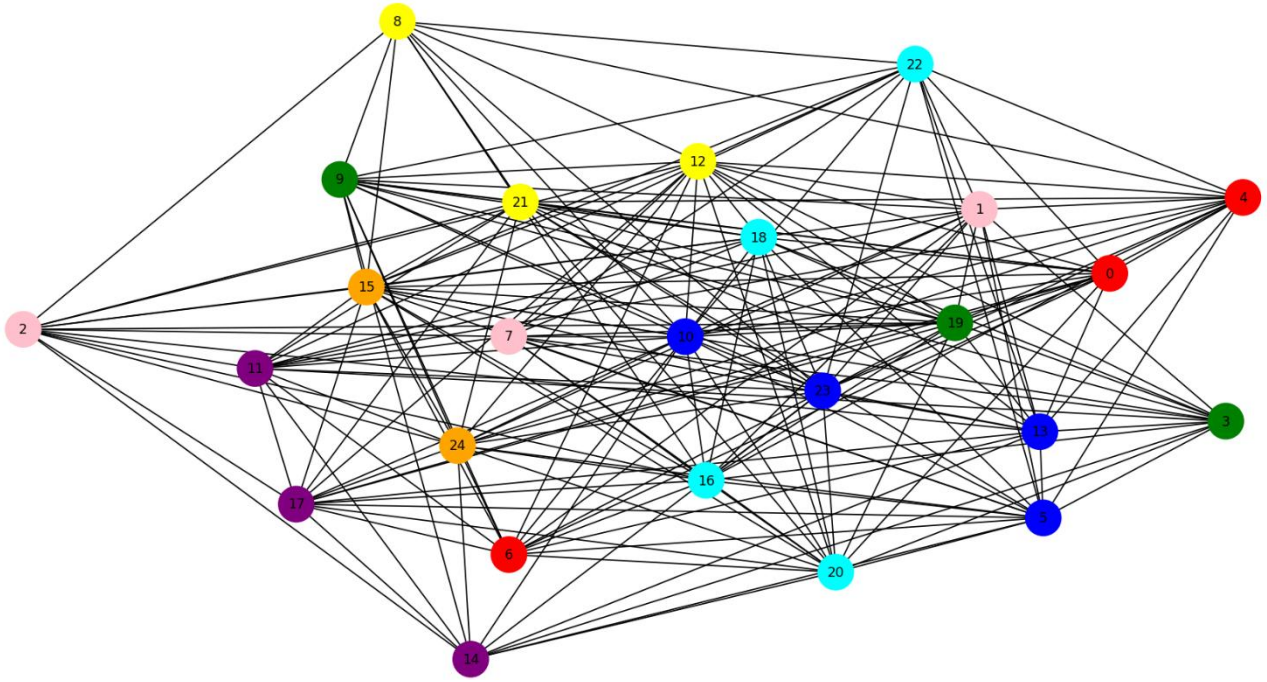
- Para cada vértice k (do 1 ao N-1):
- Cria um vetor available[] que indica quais cores estão disponíveis.
- Percorre todos os vizinhos de k e marca como indisponíveis (false) as cores já usadas pelos vizinhos.
- Atribui ao vértice k a menor cor disponível.

```c
void guloso() {
    for (int i = 0; i < N; i++) {
        color[i] = -1;
    }
    color[0] = 0;

    int available[N];
    for (int k = 1; k < N; k++) {
        for (int i = 0; i < N; i++) {
            available[i] = 1;
        }

        for (int i = 0; i < N; i++) {
            if (graph[k][i] && color[i] != -1) {
                available[color[i]] = 0;
            }
        }

        int menorCor = 0;
        while(!available[menorCor] && menorCor < N) {
            menorCor++;
        }

        color[k] = menorCor;
    }

    int maxColor = 0;
    for (int i = 0; i < N; i++) {
        if (color[i] > maxColor) maxColor = color[i];
    }

    printf("\n[Guloso] Número de cores usadas: %d\n", maxColor + 1);
    for (int i = 0; i < N; i++) {
        printf("Vértice %d -> Cor %d\n", i, color[i] + 1);
    }
}
```

4. IMAGENS

Foi implementado código em Python para gerar os grafos com suas respectivas cores. O código principal, onde contém o Backtracking e o Guloso, geram um arquivo txt das cores de cada grafo, que é lido pelo arquivo em Python e gerada a imagem. Dessa forma, é possível confirmar se nenhum vértice adjacente compartilha a mesma cor e comparar facilmente os resultados do backtracking e do algoritmo guloso.

Para o algoritmo B**acktracking** (8 cores)



Para o algoritmo **Guloso** (10 cores)