UNIVERSIDADE FEDERAL DE JUIZ DE FORA

Trabalho de Teoria dos Grafos Parte 1
DCC059 - Teoria dos Grafos

Felipe Lazzarini Cunha
Luiz Alberto Werneck Silva

Juiz de Fora - MG
2025

Sumário

1 Introdução

O trabalho tem como objetivo implementar diferentes representações de grafos, aplicando conceitos de teoria dos grafos, como, por exemplo, conexidade, bipartição e árvores, em um ambiente orientado a objetos.

Link repositório: `https://github.com/Muowl/trabalho-grafos-DCC059`

2 Desenvolvimento

No desenvolvimento do código, algumas questões, como estrutura, organização e boas práticas, foram abordadas de forma a deixá-lo na melhor forma possível.

2.1 Estrutura e Organização do Código

Para melhor organização e estrutura do código, o uso de herança, encapsulamento e modularização foram fundamentais.

2.1.1 Uso de Herança

Uma classe base abstrata Grafo define a interface para as operações de grafos.

Classes derivadas (GrafoLista e GrafoMatriz) implementam diferentes estratégias de armazenamento, promovendo flexibilidade e extensibilidade.

2.1.2 Encapsulamento

Atributos privados garantem controle de acesso e evitam inconsistências nos dados. Por outro lado, métodos públicos bem definidos permitem uma interação segura com as estruturas.

2.1.3 Modularização

Cada componente (e.g., lista encadeada, grafo, main) foi implementado em arquivos separados (e.g., lista_encadeada.h, grafo.h), seguindo os princípios de coesão e separação de responsabilidades.

2.2 Implementação de Boas Práticas

2.2.1 Alocação Dinâmica em Funções

Nas funções eh_bipartido e n_conexo, foram utilizados arrays alocados com new[] para manter o estado de cada vértice (e.g., corArray e visitado). Os arrays são liberados corretamente com delete[] após o uso.

Para o caso das funções possui_articulacao e possui_ponte, foram utilizados alocação de arrays, como disc, low e parent, para armazenar dados de descoberta e menor caminho em DFS. Arrays são liberados corretamente para evitar vazamentos de memória.

A importância da alocação dinâmica em funções se da pelo fato dos arrays dinâmicos serem essenciais para tratar cenários de tamanhos variáveis na quantidade de vértices do grafo.

2.2.2 Reuso de Código

Algumas práticas foram utilizadas pela melhorar a eficiência do programa, como a de funções auxiliares, como as funções dfs_color e dfs_articulacao que evitam duplicação de lógica. Além disso, outra prática é a de estruturas genéricas, como a classe ListaEncadeada é que reutilizável em outros contextos.

2.2.3 Legibilidade

Para melhor legibilidade do código, foram utilizados nomeação clara de classes, métodos e variáveis, além de comentários explicativos em pontos críticos do código.

2.2.4 Robustez

Para melhor robustez, foram usadas verificações de Erro,que foram implementadas durante o carregamento de arquivos e outras operações críticas. Além disso, houve gerenciamento de memória através do uso correto de new e delete para evitar vazamentos.

2.2.5 Orientação a Objetos

Ao utilizar a orientação a objetos, o polimorfismo irá permitir tratar objetos de diferentes classes derivadas de forma uniforme. Além disso, com extensibilidade, métodos virtuais garantem a possibilidade de adicionar funcionalidades sem modificar a classe base.

Tem-se, então, a classe pai (grafo.h):

```
grafo.h > ...
#ifndef GRAFO_H
#define GRAFO_H

#include <string>

class Grafo {
public:
    virtual ~Grafo() = default;

    // Funções necessárias
    virtual bool eh_bipartido() const = 0;
    virtual int n_conexo() const = 0;
    virtual int get_grau(int vertice) const = 0;
    virtual int get_ordem() const = 0;
    virtual bool eh_direcionado() const = 0;
    virtual bool vertice_ponderado() const = 0;
    virtual bool aresta_ponderada() const = 0;
    virtual bool eh_completo() const = 0;
    virtual bool eh_arvore() const = 0;
    virtual bool possui_articulacao() const = 0;
    virtual bool possui_ponte() const = 0;
    virtual void carrega_grafo(const std::string& arquivo) = 0;
    virtual void novo_grafo(const std::string& descricao, const std::string& saida) = 0;
};

#endif // GRAFO_H
```

Figura 1: grafo.h

3 Adversidades

Houveram algumas adversidades no desenvolvimento do trabalho, como, por exemplo, restrições e alguns problemas que surgiram ao construir o código.

3.1 Restrições

As restrições foram colocadas no uso de bibliotecas, onde apenas bibliotecas permitidas foram utilizadas (fstream, iostream, etc.). Além disso, em lista encadeada implementação própria para substituir o uso de std::list. Ademais, requisitos em métodos como eh_bipartido, n_conexo e eh_completo foram implementados conforme solicitado. Outra restrição foi a geração de grafos aleatórios e carregamento a partir de arquivos respeitam o formato descrito no enunciado. Todas foram satisfeitas.

3.2 Problemas enfrentados

3.2.1 Alocação de Memória

Durante os testes iniciais, problemas relacionados à alocação e desalocação de memória foram detectados com o uso do Valgrind. O gerenciamento correto foi implementado posteriormente, resolvendo os erros.

3.2.2 Lógica de Funções Auxiliares

Funções como dfs_color e dfs_articulacao exigiram pesquisas adicionais para superar dificuldades iniciais na lógica.

```
// Funções auxiliares como "friend" para acessarem 'vertices'
friend bool dfs_color(const GrafoLista& g, int v, int corAtual, int* corArray);
friend void dfs_componente(const GrafoLista& g, int v, bool* visitado);
friend void dfs_articulacao(const GrafoLista& g, int v, bool* visitado, int* disc, int* low, int* parent, bool* ap,
int& time);
friend void dfs_ponte(const GrafoLista& g, int v, bool* visitado, int* disc, int* low, int* parent, bool&
has_bridge, int& time);
```

Figura 2: Funções dfs

4 Conclusão

Alguns pontos principais podem ser ressaltados na conclusão, os resultados alcançados e o algumas considerações finais do projeto.

4.1 Resultados Alcançados

Foi alcançado um código modular, reutilizável e alinhado às boas práticas de orientação a objetos. Além do fato de que, todas as funcionalidades obrigatórias foram implementadas, incluindo detecção de pontes, articulações e bipartição.

4.2 Considerações Finais

O projeto reforça a importância de boas práticas na implementação de estruturas de grafos. Na segunda parte do trabalho, espera-se expandir e melhorar o que foi criado, com tempo adicional para implementar os possíveis algoritmos futuros (como Dijkstra e Floyd).