Universidade Federal de Juiz de Fora

Trabalho de Teoria dos Grafos - 2024.3

Parte 1- Reenvio

Integrantes:

Felipe Lazzarini Cunha
Luiz Alberto Werneck Silva

10 de fevereiro de 2025

Sumário

Capítulo 1

Introdução

Apresentação feita em sala utilizando o método do elevator pitch, como devidamente pedido no roteiro do trabalho.

Link do repositório: https://github.com/Muowl/trabalho-grafos-DCC059/tree/ajustes-parte-1

Os integrantes ficaram divididos com as seguintes contribuições: Felipe responsável pelo grafo_lista e Luiz responsável pelo grafo_matriz.

O trabalho foi sendo desenvolvido e testado na ultima versão do Ubuntu (através de imagem Docker). Para execução do programa podemos usar a imagem, ou usando o CMake.

```
# Usar a imagem base do Ubuntu
FROM ubuntu:latest

# Atualizar o sistema e instalar dependências
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    g++ \
    git \
    valgrind \
    && apt-get clean

# Definir o diretório de trabalho
WORKDIR /app

# Copiar o conteúdo do projeto para o diretório de trabalho
COPY . /app

# Criar o diretório de build
RUN mkdir -p build

# Entrar no diretório de build e compilar o projeto
WORKDIR /app/build
RUN rm -rf CMakeCache.txt CMakeFiles
RUN cmake .. && make
```

Figura 1.1: Print do arquivo Dockerfile, presente na pasta raiz do trabalho.

Capítulo 2

Arquivos Auxiliares

Nesta seção, são descritas as estruturas auxiliares que foram implementadas para facilitar o desenvolvimento e o funcionamento do projeto. Em especial, abordam-se os arquivos “leitura.h” e “lista_encadeada.h”.

2.1 Arquivo “leitura.h”

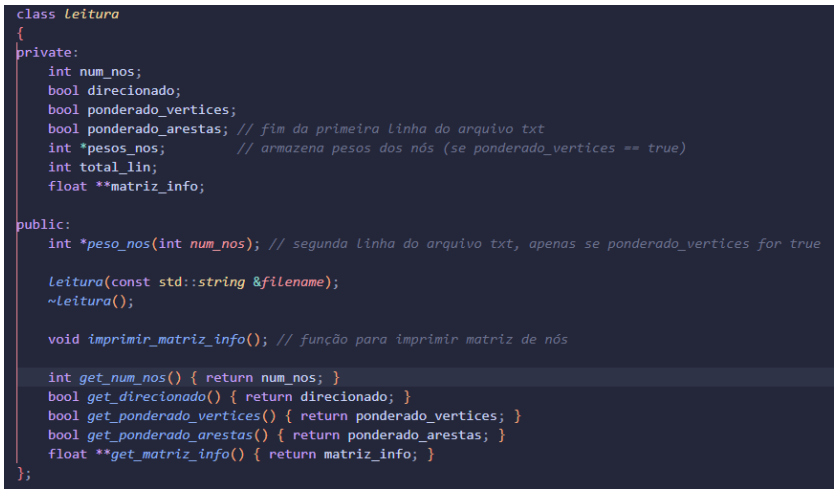

Figura 2.1: Print do arquivo leitura.h.

O arquivo “leitura.h” define a classe “leitura”, responsável por ler os dados de entrada a partir de um arquivo de texto e estruturar essas informações em variáveis que poderão ser utilizadas para a criação dos grafos. A classe possui os seguintes membros principais:

- **num_nos**: número de nós (vértices) contidos no grafo.

- **direcionado**: variável booleana que indica se o grafo é direcionado.
- **ponderado_vertices** e **ponderado_arestas**: indicam se os nós e as arestas possuem pesos, respectivamente.
- **pesos_nos**: vetor que armazena os pesos dos nós, caso "ponderado_vertices" seja verdadeiro.
- **matriz_info**: matriz dinâmica que armazena a relação entre os nós e as arestas (usada para representar cada linha do arquivo, contendo origem, destino e peso).

A classe também fornece funções auxiliares, como o método "imprimir_matriz_info()", que exibe no console a estrutura lida do arquivo, e diversos getters para acessar os atributos. Assim, "leitura.h" é uma peça fundamental para a separação da lógica de entrada de dados, permitindo que as classes representativas dos grafos (como as utilizadas em "grafo.h" e suas derivações) possam se concentrar na implementação de operações e algoritmos sobre os grafos, sem se preocupar com o processo de leitura e validação dos dados.

2.2 Arquivo "lista_encadeada.h"

O arquivo "lista_encadeada.h" define uma classe template para uma lista encadeada, uma estrutura de dados genérica que permite armazenar elementos de qualquer tipo. Esta implementação de lista encadeada possui características essenciais:

- Gerenciamento dinâmico dos nós, possibilitando a inserção de novos elementos com a função "push_back()".
- Funções utilitárias, como "clear()", "get_size()" e "empty()", que permitem o gerenciamento e a verificação do estado da lista.
- Iteradores (e iteradores constantes) que facilitam a navegação pelos elementos armazenados na lista.

Esta estrutura é especialmente útil na implementação de grafos baseados em listas (como em "grafo_lista.h"), onde é necessário armazenar vértices e arestas de forma dinâmica e eficiente. A flexibilidade oferecida pelos templates permite reutilizar a mesma estrutura para diferentes tipos de dados, promovendo reuso de código e melhor organização do projeto.

```
#ifndef LISTA_ENCADEADA_H
#define LISTA_ENCADEADA_H

template <typename T>
class ListaEncadeada {
private:
    struct Node {
        T data;
        Node* next;
        Node(const T& d) : data(d), next(nullptr) {}
    };

    Node* head;
    Node* tail;
    int size;

public:
    ListaEncadeada() : head(nullptr), tail(nullptr), size(0) {}
    ~ListaEncadeada();

    void push_back(const T& value);
    void clear();
    int get_size() const;
    bool empty() const;

    class Iterator {
    private:
        Node* current;
    public:
        Iterator(Node* node) : current(node) {}
        T& operator*() { return current->data; }
        T* operator->() { return &current->data; }
```

Figura 2.2: Print do arquivo “lista_encadeada.h.”

2.3 Em resumo

Ao separar a funcionalidade de leitura de dados e a implementação de estruturas de dados dinâmicas, o projeto ganha modularidade, facilitando a manutenção e evolução do código. Especificamente:

- A classe “leitura” abstrai o trabalho de interpretar o arquivo de entrada, permitindo que futuras modificações na formatação do arquivo sejam concentradas nesta classe, sem impactar outras partes do sistema.
- A classe “ListaEncadeada” fornece uma implementação genérica de listas que pode ser utilizada em diversas partes do código, como por exemplo na representação dos vértices e arestas em “grafo_lista”.

Essa abordagem modular promove também a extensibilidade do projeto, permitindo a criação de novas funcionalidades (por exemplo, novas estruturas de grafos) sem a necessidade de modificar as partes já testadas e validadas do código.

Com estas implementações, os arquivos auxiliares desempenham um papel fundamental na organização do projeto, garantindo que cada parte da aplicação seja responsável por uma funcionalidade específica, contribuindo para um design mais limpo e robusto, e preservando os preceitos de orientação a objetos.

Capítulo 3

Arquivos Principais

Nesta seção, apresentamos a implementação dos principais arquivos responsáveis pela manipulação dos grafos.

3.1 grafo.h

O arquivo `grafo.h` define a classe abstrata base para a representação dos grafos. Nele, são estabelecidos os atributos fundamentais, como a ordem (número de vértices), se o grafo é direcionado, e se os vértices e arestas possuem pesos. Além disso, métodos básicos como `get_ordem()`, `eh_completo()` e `carrega_grafo()` são declarados, permitindo que classes derivadas implementem suas funcionalidades específicas.

3.2 grafo_lista

A implementação contida em `grafo_lista.h` deriva da classe `grafo` e utiliza uma lista encadeada para armazenar os vértices e as arestas. Essa estrutura permite a manipulação dinâmica dos elementos do grafo. Métodos específicos, como `get_grau()` e `n_conexo()`, foram implementados para calcular propriedades importantes do grafo. A função `carrega_grafo()` utiliza a classe `leitura` para carregar os dados do arquivo de entrada e inicializar os atributos do grafo. Essa abordagem modular facilita a extensão e manutenção do sistema, mantendo a separação entre a lógica abstrata e a implementação concreta.

3.3 grafo_matriz

A implementação contida em `grafo_matriz.h` também deriva da classe `grafo`, mas utiliza uma matriz de adjacência para armazenar os vértices e as arestas de forma estática. Essa estrutura permite a manipulação eficiente dos elementos do grafo, especialmente para operações que envolvem a verificação de conexões entre vértices. Métodos específicos, como `get_grau()` e `n_conexo()`, foram implementados para calcular propriedades importantes do grafo. A função `carrega_grafo()` utiliza a classe `leitura` para carregar os dados do arquivo de entrada e inicializar os atributos do grafo. A matriz de adjacência é inicializada e preenchida com as arestas lidas do arquivo. Essa abordagem modular facilita a extensão e manutenção do sistema, mantendo a separação entre a lógica abstrata e a implementação concreta.

Capítulo 4

Considerações Finais

Ao final do trabalho, foi feito o teste de memória com o Valgrid e o seguinte resultado foi obtido:

```
==7== Memcheck, a memory error detector
==7== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==7== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==7== Command: ./main -d -m ../entradas/grafo.txt
==7== Parent PID: 1
==7==
==7==
==7== HEAP SUMMARY:
==7==     in use at exit: 0 bytes in 0 blocks
==7==   total heap usage: 17 allocs, 17 frees, 94,284 bytes allocated
==7==
==7== All heap blocks were freed -- no leaks are possible
==7==
==7== For lists of detected and suppressed errors, rerun with: -s
==7== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
-e
```

Figura 4.1: Resultado Valgrid

Como pode ser visto na figura, não houve erros e todos os espaços foram alocados devidamente, não obtendo vazamentos de memória.