Universidade Federal de Juiz de Fora

Trabalho de Teoria dos Grafos - 2024.3

Parte 2

Integrantes:

Felipe Lazzarini Cunha
Luiz Alberto Werneck Silva

11 de fevereiro de 2025

Sumário

Capítulo 1

Introdução

O capítulo 3 dos "Arquivos Principais"aborda os temas relacionados a parte 2 do trabalho. Para explicações específicas das funções da parte 1 consultar a documentação Parte 1, presente no diretório "/docs"no repositório GitHub.
Link do repositório: https://github.com/Muowl/trabalho-grafos-DCC059/tree/parte-2
Os integrantes ficaram divididos com as seguintes contribuições: Felipe responsável pelo grafo_lista e Luiz responsável pelo grafo_matriz.
Será apresentado no estilo elevator pitch, como requisitado na descrição parte 2, no capítulo "Da apresentação".
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”

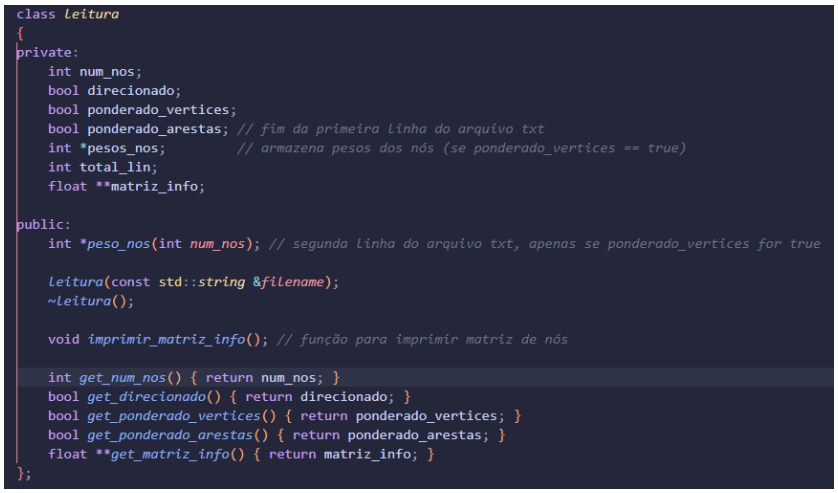

Figura 2.1: Print do arquivo leitura.h.

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 das funções respectivas da parte 2.

3.1 "grafo_lista"

- **`novo_no(float peso)`**: Cria e adiciona um novo vértice ao grafo com o próximo *id* disponível e o peso informado. Em seguida, atualiza o atributo *ordem* para refletir o acréscimo de um novo vértice.

- **`nova_aresta(int origem, int destino, float peso)`**: Cria uma nova aresta entre dois vértices válidos, recebendo *id* de origem, *id* de destino e peso. Caso o grafo seja não direcionado, a aresta é considerada bidirecional, e a função adiciona automaticamente a aresta reversa (`destino→origem`).

- **`deleta_no(int id)`**: Remove um vértice do grafo, eliminando também todas as arestas que o referenciem como origem ou destino. O armazenamento interno do grafo (lista de nós e lista de arestas) é atualizado para suprimir esse vértice, e a *ordem* do grafo é decrementada. Após a remoção, os IDs dos vértices restantes são reindexados para garantir que sejam sequenciais e estejam no intervalo `[0, ordem-1]`. Isso inclui a atualização das referências nas arestas para refletir os novos IDs.

- **`deleta_aresta(int origem, int destino)`**: Exclui a primeira aresta que corresponda aos valores de *origem* e *destino* passados como parâmetro. Em grafos não direcionados, a função também remove a aresta reversa (`destino→origem`) para manter a consistência da estrutura.

- **`menor_maior_distancia()`**: Calcula a maior distância mínima entre dois vértices no grafo usando o algoritmo *Floyd-Warshall*. Retorna uma estrutura com:
  - `no1`: ID do primeiro vértice (convertido para 1-based),
  - `no2`: ID do segundo vértice (convertido para 1-based),
  - `distancia`: Valor da maior distância mínima encontrada.

  Ignora pares desconectados e prioriza arestas de menor peso em casos de múltiplas conexões.

Estas cinco funções complementam a classe `grafo_lista`, fornecendo suporte para manipulações dinâmicas nos vértices e arestas. Dessa forma, é possível inserir ou remover elementos do grafo em tempo de execução, mantendo a estrutura de dados totalmente atualizada e consistente.

3.2 "grafo_matriz"

- **`novo_no(float peso)`**: Invoca a função interna `add_no` para criar um novo nó com o peso informado, atualizando a estrutura interna (vetor de nós e a ordem do grafo). Isso garante que o novo nó receba o próximo *id* disponível.
- **`nova_aresta(int origem, int destino, float peso)`**: Chama a função `add_aresta`, que converte os índices de 1-based para 0-based. A função adiciona a aresta na matriz de adjacência com o peso especificado e, em grafos não direcionados, atualiza a posição simétrica.
- **`deleta_no(int id)`**: Remove o nó indicado (convertendo de 1-based para 0-based) e todas as arestas associadas a ele. Após a remoção, realiza o *shift* dos elementos na matriz de adjacência e no vetor de nós, reindexando os *ids* dos nós restantes e decrementando a ordem do grafo.
- **`deleta_aresta(int origem, int destino)`**: Exclui uma aresta entre os nós indicados (convertendo os índices de 1-based para 0-based). Em grafos não direcionados, também remove a aresta no sentido inverso para manter a consistência na matriz de adjacência.
- **`menor_maior_distancia()` e sua `struct MenorMaior`**:
  - A `struct MenorMaior` contém os campos:

    - `no1`: *id* do primeiro nó (em 1-based) do par com a maior distância mínima.
    - `no2`: *id* do segundo nó (em 1-based).
    - `distancia`: valor da distância mínima entre estes nós.
- A função `menor_maior_distancia` utiliza o algoritmo de Floyd–Warshall para calcular os caminhos mínimos entre todos os pares de nós. Ela inicializa uma matriz auxiliar de distâncias (usando um valor alto para representar o infinito), atualiza as distâncias considerando os caminhos intermediários e, por fim, percorre essa matriz para identificar o par com a maior distância mínima (desde que seja finita). O resultado é retornado em uma instância da `struct MenorMaior`, com os *ids* convertidos para 1-based.

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.