Teoria dos Grafos

Generated by Doxygen 1.13.2

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 ArestaEncadeada Class Reference

`#include <ArestaEncadeada.h>`

Public Member Functions

- ArestaEncadeada (VerticeEncadeado *origem, VerticeEncadeado *destino, float peso)
- VerticeEncadeado * getOrigem () const
- VerticeEncadeado * getDestino () const
- float getPeso () const
- ArestaEncadeada * getProximo () const
- void setProximo (ArestaEncadeada *novoProximo)

Friends

- std::ostream & operator<< (std::ostream &os, const ArestaEncadeada &aresta)

4.1.1 Constructor & Destructor Documentation

4.1.1.1 ArestaEncadeada()

```
ArestaEncadeada::ArestaEncadeada (
            VerticeEncadeado * origem,
            VerticeEncadeado * destino,
            float peso)
```

4.1.2 Member Function Documentation

4.1.2.1 getDestino()

```
VerticeEncadeado * ArestaEncadeada::getDestino () const
```

4.1.2.2 getOrigem()

```
VerticeEncadeado * ArestaEncadeada::getOrigem () const
```

4.1.2.3 getPeso()

```
float ArestaEncadeada::getPeso () const
```

4.1.2.4 getProximo()

```
ArestaEncadeada * ArestaEncadeada::getProximo () const
```

4.1.2.5 setProximo()

```
void ArestaEncadeada::setProximo (
            ArestaEncadeada * novoProximo)
```

4.1.3 Friends And Related Symbol Documentation

4.1.3.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
            const ArestaEncadeada & aresta)  [friend]
```

The documentation for this class was generated from the following files:

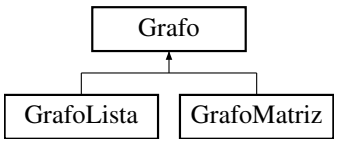
- include/ArestaEncadeada.h
- src/ArestaEncadeada.cpp

4.2 Grafo Class Reference

Classe base para a representação de um grafo.

```
#include <Grafo.h>
```

Inheritance diagram for Grafo:



Public Member Functions

- Grafo ()=default

  Construtor padrão.
- void carrega_grafo_novo (int d)

  Carrega o grafo a partir de um arquivo que contem a instancia do grafo.
- virtual ~Grafo ()

  Destruidor virtual.
- virtual int get_aresta (int origem, int destino)=0

  Método abstrato para obter o peso de uma aresta entre dois vértices.
- virtual int get_vertice (int vertice)=0

  Método abstrato para obter o peso de um vértice.
- virtual int get_vizinhos (int vertice)=0

  Método abstrato para obter o número de vizinhos de um vértice.
- virtual void nova_aresta (int origem, int destino, int peso)=0

  Método abstrato para adicionar uma nova aresta entre dois vértices.
- virtual void set_aresta (int origem, int destino, float peso)=0

  Método abstrato para definir o peso de uma aresta.
- virtual void set_vertice (int id, float peso)=0

  Método abstrato para definir o peso de um vértice.
- virtual int * get_vizinhos_vertices (int vertice, int &qtdvizinhos)=0

  Obtém os vértices vizinhos de um determinado vértice.
- virtual int * get_vizinhos_array (int id, int &tamanho)=0
- int get_ordem ()

  Obtém o número de vértices no grafo (ordem do grafo).
- void set_ordem (int ordem)

  Define o número de vértices do grafo.
- void aumenta_ordem ()

  Aumenta a ordem do grafo em 1.
- bool eh_direcionado ()

  Verifica se o grafo é direcionado.
- void set_eh_direcionado (bool direcionado)

  Define se o grafo é direcionado.
- bool vertice_ponderado ()

  Verifica se os vértices do grafo são ponderados.
- void set_vertice_ponderado (bool verticePonderado)

  Define se os vértices do grafo são ponderados.
- bool aresta_ponderada ()

  Verifica se as arestas do grafo são ponderadas.
- void set_aresta_ponderada (bool arestaPonderada)

  Define se as arestas do grafo são ponderadas.
- int get_grau ()

  Obtém o grau (número de vizinhos) do vértice com maior grau.
- bool eh_completo ()

  Verifica se o grafo é completo.
- void dfs (int vertice, bool visitado[])

  Realiza uma busca em profundidade (DFS) para explorar todos os vértices conectados a partir de um vértice inicial.
- int n_conexo ()

  Calcula o número de componentes conexas do grafo.
- virtual void inicializa_grafo ()=0

  Método abstrato para inicializar o grafo, implementado pelas classes derivadas.

- void maior_menor_distancia ()

  Encontra a maior menor distância entre os vértices utilizando o algoritmo de Floyd-Warshall.
- void criar_clusters_aleatorios (int num_clusters)
- int * get_clusters ()
- float encontrar_agmg_guloso ()

  Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.
- float encontrar_agmg_randomizado ()

  Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.
- void testa_get_vizinhos (int id)

4.2.1 Detailed Description

Classe base para a representação de um grafo.

Essa classe define as operações gerais que podem ser realizadas em grafos, como manipulação de arestas, vértices, grau, e conexões. As subclasses precisam implementar a inicialização do grafo e operações de manipulação específicas.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Grafo()

```
Grafo::Grafo () [default]
```

Construtor padrão.

4.2.2.2 ~Grafo()

```
virtual Grafo::~Grafo () [inline], [virtual]
```

Destruidor virtual.

4.2.3 Member Function Documentation

4.2.3.1 aresta_ponderada()

```
bool Grafo::aresta_ponderada () [inline]
```

Verifica se as arestas do grafo são ponderadas.

Returns

Verdadeiro se as arestas forem ponderadas, falso caso contrário.

4.2.3.2 aumenta_ordem()

```
void Grafo::aumenta_ordem () [inline]
```

Aumenta a ordem do grafo em 1.

4.2.3.3 carrega_grafo_novo()

```
void Grafo::carrega_grafo_novo (
            int d) [inline]
```

Carrega o grafo a partir de um arquivo que contem a instancia do grafo.

Parameters

d	Parametro que seleciona o arquivo a ser lido

4.2.3.4 criar_clusters_aleatorios()

```
void Grafo::criar_clusters_aleatorios (
            int num_clusters)  [inline]
```

4.2.3.5 dfs()

```
void Grafo::dfs (
            int vertice,
            bool visitado[])  [inline]
```

Realiza uma busca em profundidade (DFS) para explorar todos os vértices conectados a partir de um vértice inicial.

Parameters

vertice	Vértice de origem para a busca.
visitado	Array de visitados para marcar os vértices já visitados.

4.2.3.6 eh_completo()

```
bool Grafo::eh_completo ()  [inline]
```

Verifica se o grafo é completo.

Returns

Verdadeiro se o grafo for completo (todos os vértices estão conectados entre si), falso caso contrário.

4.2.3.7 eh_direcionado()

```
bool Grafo::eh_direcionado ()  [inline]
```

Verifica se o grafo é direcionado.

Returns

Verdadeiro se o grafo for direcionado, falso caso contrário.

4.2.3.8 encontrar_agmg_guloso()

```
float Grafo::encontrar_agmg_guloso ()  [inline]
```

Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.

Esta função utiliza uma abordagem gulosa

Returns

O somatorio dos pesos das arestas que compoe a arvore

4.2.3.9 encontrar_agmg_randomizado()

```
float Grafo::encontrar_agmg_randomizado ()  [inline]
```

Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.

Esta função utiliza uma abordagem gulosa randomizada

Returns

O somatorio dos pesos das arestas que compoe a arvore

4.2.3.10 get_aresta()

```
virtual int Grafo::get_aresta (
            int origem,
            int destino)  [pure virtual]
```

Método abstrato para obter o peso de uma aresta entre dois vértices.

Parameters

origem	Vértice de origem da aresta.
destino	Vértice de destino da aresta.

Returns

O peso da aresta.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.11 get_clusters()

```
int * Grafo::get_clusters ()  [inline]
```

4.2.3.12 get_grau()

```
int Grafo::get_grau ()  [inline]
```

Obtém o grau (número de vizinhos) do vértice com maior grau.

Returns

O grau máximo.

4.2.3.13 get_ordem()

```
int Grafo::get_ordem ()  [inline]
```

Obtém o número de vértices no grafo (ordem do grafo).

Returns

O número de vértices do grafo.

4.2.3.14 get_vertice()

```
virtual int Grafo::get_vertice (
            int vertice)  [pure virtual]
```

Método abstrato para obter o peso de um vértice.

Parameters

vertice	O identificador do vértice.

Returns

O peso do vértice.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.15 get_vizinhos()

```
virtual int Grafo::get_vizinhos (
            int vertice)  [pure virtual]
```

Método abstrato para obter o número de vizinhos de um vértice.

Parameters

vertice	O identificador do vértice.

Returns

O número de vizinhos.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.16 get_vizinhos_array()

```
virtual int * Grafo::get_vizinhos_array (
            int id,
            int & tamanho)  [pure virtual]
```

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.17 get_vizinhos_vertices()

```
virtual int * Grafo::get_vizinhos_vertices (
            int vertice,
            int & qtdvizinhos)  [pure virtual]
```

Obtém os vértices vizinhos de um determinado vértice.

Esta função retorna um array contendo os vértices vizinhos do vértice especificado. A quantidade de vizinhos é armazenada na variável passada por referência.

Parameters

vertice	O vértice cujo vizinhos serão buscados.
qtdvizinhos	Referência para armazenar a quantidade total de vizinhos.

Returns

Ponteiro para um array contendo os vértices vizinhos.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.18 inicializa_grafo()

```
virtual void Grafo::inicializa_grafo ()  [pure virtual]
```

Método abstrato para inicializar o grafo, implementado pelas classes derivadas.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.19 maior_menor_distancia()

```
void Grafo::maior_menor_distancia ()  [inline]
```

Encontra a maior menor distância entre os vértices utilizando o algoritmo de Floyd-Warshall.

O algoritmo calcula todas as menores distâncias entre pares de vértices e depois encontra a maior distância entre qualquer par de vértices. Caso não haja caminho entre os vértices, o algoritmo considerará um valor de infinito para aquelas distâncias.

4.2.3.20 n_conexo()

```
int Grafo::n_conexo ()  [inline]
```

Calcula o número de componentes conexas do grafo.

Returns

O número de componentes conexas no grafo.

4.2.3.21 nova_aresta()

```
virtual void Grafo::nova_aresta (
            int origem,
            int destino,
            int peso)  [pure virtual]
```

Método abstrato para adicionar uma nova aresta entre dois vértices.

Parameters

origem	Vértice de origem da aresta.
destino	Vértice de destino da aresta.
peso	Peso da aresta.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.22 set_aresta()

```
virtual void Grafo::set_aresta (
            int origem,
            int destino,
            float peso)  [pure virtual]
```

Método abstrato para definir o peso de uma aresta.

Parameters

origem	Vértice de origem da aresta.
destino	Vértice de destino da aresta.
peso	Peso da aresta.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.23 set_aresta_ponderada()

```
void Grafo::set_aresta_ponderada (
            bool arestaPonderada)  [inline]
```

Define se as arestas do grafo são ponderadas.

Parameters

arestaPonderada	Valor que define se as arestas serão ponderadas.

4.2.3.24 set_eh_direcionado()

```
void Grafo::set_eh_direcionado (
            bool direcionado)  [inline]
```

Define se o grafo é direcionado.

Parameters

direcionado	Valor que define se o grafo será direcionado.

4.2.3.25 set_ordem()

```
void Grafo::set_ordem (
            int ordem)  [inline]
```

Define o número de vértices do grafo.

Parameters

ordem	O número de vértices a ser definido.

4.2.3.26 set_vertice()

```
virtual void Grafo::set_vertice (
            int id,
            float peso)  [pure virtual]
```

Método abstrato para definir o peso de um vértice.

Parameters

id	Identificador do vértice.
peso	Peso do vértice.

Implemented in GrafoLista, and GrafoMatriz.

4.2.3.27 set_vertice_ponderado()

```
void Grafo::set_vertice_ponderado (
            bool verticePonderado)  [inline]
```

Define se os vértices do grafo são ponderados.

Parameters

verticePonderado	Valor que define se os vértices serão ponderados.

4.2.3.28 testa_get_vizinhos()

```
void Grafo::testa_get_vizinhos (
            int id)  [inline]
```

4.2.3.29 vertice_ponderado()

```
bool Grafo::vertice_ponderado ()  [inline]
```

Verifica se os vértices do grafo são ponderados.

Returns

Verdadeiro se os vértices forem ponderados, falso caso contrário.

The documentation for this class was generated from the following file:

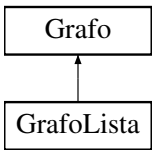
- include/Grafo.h

4.3 GrafoLista Class Reference

A classe GrafoLista é uma implementação de grafo que utiliza listas encadeadas para armazenar os vértices e arestas. Ela herda da classe abstrata Grafo e implementa suas funções virtuais para manipulação de vértices e arestas.

```
#include <GrafoLista.h>
```

Inheritance diagram for GrafoLista:



Public Member Functions

- GrafoLista ()

  Construtor da classe GrafoLista. Inicializa as listas de vértices e arestas.
- int get_vertice (int id) override

  Método para obter o peso de um vértice dado seu id.
- int get_aresta (int idOrigem, int idDestino) override

  Método para obter o peso de uma aresta entre dois vértices dados seus ids.
- void set_vertice (int id, float peso) override

  Método para definir o peso de um vértice dado seu id.
- void set_aresta (int origem, int destino, float peso) override

  Método para definir o peso de uma aresta entre dois vértices dados seus ids.
- void nova_aresta (int origem, int destino, int peso) override

  Método para adicionar uma nova aresta entre dois vértices. Verifica se a aresta já existe e, caso contrário, adiciona a aresta na lista.
- int get_vizinhos (int vertice) override

  Método para obter o número de vizinhos de um vértice.
- void imprimir ()

  Método para imprimir os vértices e as arestas do grafo, além de informações sobre o grau e componentes conexas.
- void inicializa_grafo () override

  Método para inicializar o grafo a partir de um arquivo de entrada. Lê as informações de vértices, arestas, pesos e outras configurações.
- int * get_vizinhos_array (int id, int &tamanho) override
- int * get_vizinhos_vertices (int vertice, int &quantidadeVizinhos) override

  Obtém os vértices vizinhos de um determinado vértice.
- ~GrafoLista ()

  Destruidor da classe GrafoLista. Libera a memória alocada para as listas de vértices e arestas.

Public Member Functions inherited from Grafo

- Grafo ()=default

  Construtor padrão.
- void carrega_grafo_novo (int d)

  Carrega o grafo a partir de um arquivo que contem a instancia do grafo.
- virtual ~Grafo ()

  Destruidor virtual.
- int get_ordem ()

  Obtém o número de vértices no grafo (ordem do grafo).
- void set_ordem (int ordem)

  Define o número de vértices do grafo.
- void aumenta_ordem ()

  Aumenta a ordem do grafo em 1.
- bool eh_direcionado ()

  Verifica se o grafo é direcionado.
- void set_eh_direcionado (bool direcionado)

  Define se o grafo é direcionado.
- bool vertice_ponderado ()

  Verifica se os vértices do grafo são ponderados.
- void set_vertice_ponderado (bool verticePonderado)

  Define se os vértices do grafo são ponderados.
- bool aresta_ponderada ()

Verifica se as arestas do grafo são ponderadas.

- void set_aresta_ponderada (bool arestaPonderada)

  Define se as arestas do grafo são ponderadas.
- int get_grau ()

  Obtém o grau (número de vizinhos) do vértice com maior grau.
- bool eh_completo ()

  Verifica se o grafo é completo.
- void dfs (int vertice, bool visitado[ ])

  Realiza uma busca em profundidade (DFS) para explorar todos os vértices conectados a partir de um vértice inicial.
- int n_conexo ()

  Calcula o número de componentes conexas do grafo.
- void maior_menor_distancia ()

  Encontra a maior menor distância entre os vértices utilizando o algoritmo de Floyd-Warshall.
- void criar_clusters_aleatorios (int num_clusters)
- int ∗ get_clusters ()
- float encontrar_agmg_guloso ()

  Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.
- float encontrar_agmg_randomizado ()

  Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.
- void testa_get_vizinhos (int id)

4.3.1 Detailed Description

A classe GrafoLista é uma implementação de grafo que utiliza listas encadeadas para armazenar os vértices e arestas. Ela herda da classe abstrata Grafo e implementa suas funções virtuais para manipulação de vértices e arestas.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 GrafoLista()

```
GrafoLista::GrafoLista ()
```

Construtor da classe GrafoLista. Inicializa as listas de vértices e arestas.

4.3.2.2 ∼GrafoLista()

```
GrafoLista::~GrafoLista ()
```

Destruidor da classe GrafoLista. Libera a memória alocada para as listas de vértices e arestas.

4.3.3 Member Function Documentation

4.3.3.1 get_aresta()

```
int GrafoLista::get_aresta (
            int idOrigem,
            int idDestino)  [override], [virtual]
```

Método para obter o peso de uma aresta entre dois vértices dados seus ids.

Parameters

idOrigem	O id do vértice de origem.
idDestino	O id do vértice de destino.

Returns

O peso da aresta entre os vértices de origem e destino.

Implements Grafo.

4.3.3.2 get_vertice()

```
int GrafoLista::get_vertice (
            int id)  [override], [virtual]
```

Método para obter o peso de um vértice dado seu id.

Parameters

id	O id do vértice.

Returns

O peso do vértice correspondente ao id.

Implements Grafo.

4.3.3.3 get_vizinhos()

```
int GrafoLista::get_vizinhos (
            int vertice)  [override], [virtual]
```

Método para obter o número de vizinhos de um vértice.

Parameters

vertice	O id do vértice.

Returns

O número de vizinhos do vértice.

Implements Grafo.

4.3.3.4 get_vizinhos_array()

```
int * GrafoLista::get_vizinhos_array (
            int id,
            int & tamanho)  [override], [virtual]
```

Implements Grafo.

4.3.3.5 get_vizinhos_vertices()

```
int * GrafoLista::get_vizinhos_vertices (
            int vertice,
            int & quantidadeVizinhos)  [override], [virtual]
```

Obtém os vértices vizinhos de um determinado vértice.

Esta função retorna um array contendo os vértices vizinhos do vértice especificado. A quantidade de vizinhos é armazenada na variável passada por referência.

Parameters

vertice	O vértice cujo vizinhos serão buscados.
quantidadeVizinhos	Referência para armazenar a quantidade total de vizinhos.

Returns

Ponteiro para um array contendo os vértices vizinhos.

Implements Grafo.

4.3.3.6 imprimir()

```
void GrafoLista::imprimir ()
```

Método para imprimir os vértices e as arestas do grafo, além de informações sobre o grau e componentes conexas.

4.3.3.7 inicializa_grafo()

```
void GrafoLista::inicializa_grafo ()  [override], [virtual]
```

Método para inicializar o grafo a partir de um arquivo de entrada. Lê as informações de vértices, arestas, pesos e outras configurações.

Implements Grafo.

4.3.3.8 nova_aresta()

```
void GrafoLista::nova_aresta (
            int origem,
            int destino,
            int peso)  [override], [virtual]
```

Método para adicionar uma nova aresta entre dois vértices. Verifica se a aresta já existe e, caso contrário, adiciona a aresta na lista.

Parameters

origem	O id do vértice de origem.
destino	O id do vértice de destino.
peso	O peso da nova aresta.

Implements Grafo.

4.3.3.9 set_aresta()

```
void GrafoLista::set_aresta (
            int origem,
            int destino,
            float peso)  [override], [virtual]
```

Método para definir o peso de uma aresta entre dois vértices dados seus ids.

Parameters

origem	O id do vértice de origem.
destino	O id do vértice de destino.
peso	O peso a ser atribuído à aresta.

Implements Grafo.

4.3.3.10 set_vertice()

```
void GrafoLista::set_vertice (
            int id,
            float peso)  [override], [virtual]
```

Método para definir o peso de um vértice dado seu id.

Parameters

id	O id do vértice.
peso	O peso a ser atribuído ao vértice.

Implements Grafo.

The documentation for this class was generated from the following files:

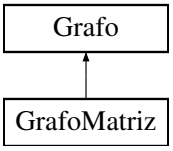
- include/GrafoLista.h
- src/GrafoLista.cpp

4.4 GrafoMatriz Class Reference

A classe GrafoMatriz implementa a interface da classe abstrata Grafo utilizando uma matriz de adjacência. A classe gerencia tanto grafos direcionados quanto não direcionados, além de permitir a manipulação de pesos de vértices e arestas.

`#include <GrafoMatriz.h>`

Inheritance diagram for GrafoMatriz:



Public Member Functions

- GrafoMatriz ()

  Construtor da classe GrafoMatriz. Inicializa as matrizes e vetores necessários.
- virtual ~GrafoMatriz ()

  Destruidor da classe GrafoMatriz. Libera a memória alocada para as matrizes e vetores.
- void redimensionarMatriz ()

  Método para redimensionar a matriz 2D de adjacência.
- void redimensionarMatrizLinear ()

  Método para redimensionar a matriz linear de adjacência.
- void inicializa_grafo ()

  Método para inicializar o grafo a partir de um arquivo de entrada. Lê os vértices, arestas e pesos do arquivo de dados.
- int calcularIndiceLinear (int origem, int destino)

  Calcula o índice linear na matriz comprimida para grafos não direcionados.
- int get_aresta (int origem, int destino) override

  Método para obter o peso de uma aresta entre dois vértices dados seus ids.
- int get_vertice (int vertice) override

  Método para obter o peso de um vértice dado seu id.
- int get_vizinhos (int vertice) override

  Método para obter o número de vizinhos de um vértice.
- void set_vertice (int id, float peso) override

  Método para definir o peso de um vértice dado seu id.
- void set_aresta (int origem, int destino, float peso) override

  Método para definir o peso de uma aresta entre dois vértices dados seus ids.
- void nova_aresta (int origem, int destino, int peso)

  Método para adicionar uma nova aresta entre dois vértices, verificando se a aresta já existe.
- int * get_vizinhos_array (int id, int &tamanho) override
- int * get_vizinhos_vertices (int vertice, int &quantidadeVizinhos) override

  Obtém os vértices vizinhos de um determinado vértice.

Public Member Functions inherited from Grafo

- Grafo ()=default

  Construtor padrão.
- void carrega_grafo_novo (int d)

  Carrega o grafo a partir de um arquivo que contem a instancia do grafo.
- virtual ~Grafo ()

  Destruidor virtual.
- int get_ordem ()

  Obtém o número de vértices no grafo (ordem do grafo).
- void set_ordem (int ordem)

  Define o número de vértices do grafo.
- void aumenta_ordem ()

  Aumenta a ordem do grafo em 1.
- bool eh_direcionado ()

  Verifica se o grafo é direcionado.
- void set_eh_direcionado (bool direcionado)

  Define se o grafo é direcionado.
- bool vertice_ponderado ()

  Verifica se os vértices do grafo são ponderados.
- void set_vertice_ponderado (bool verticePonderado)

  Define se os vértices do grafo são ponderados.
- bool aresta_ponderada ()

  Verifica se as arestas do grafo são ponderadas.
- void set_aresta_ponderada (bool arestaPonderada)

  Define se as arestas do grafo são ponderadas.
- int get_grau ()

  Obtém o grau (número de vizinhos) do vértice com maior grau.
- bool eh_completo ()

  Verifica se o grafo é completo.
- void dfs (int vertice, bool visitado[ ])

  Realiza uma busca em profundidade (DFS) para explorar todos os vértices conectados a partir de um vértice inicial.
- int n_conexo ()

  Calcula o número de componentes conexas do grafo.
- void maior_menor_distancia ()

  Encontra a maior menor distância entre os vértices utilizando o algoritmo de Floyd-Warshall.
- void criar_clusters_aleatorios (int num_clusters)
- int * get_clusters ()
- float encontrar_agmg_guloso ()

  Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.
- float encontrar_agmg_randomizado ()

  Encontra o Árvore Geradora Mínima (AGM) do grafo, garantindo que contenha pelo menos um vértice de cada cluster.
- void testa_get_vizinhos (int id)

4.4.1 Detailed Description

A classe GrafoMatriz implementa a interface da classe abstrata Grafo utilizando uma matriz de adjacência. A classe gerencia tanto grafos direcionados quanto não direcionados, além de permitir a manipulação de pesos de vértices e arestas.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 GrafoMatriz()

```
GrafoMatriz::GrafoMatriz ()
```

Construtor da classe GrafoMatriz. Inicializa as matrizes e vetores necessários.

4.4.2.2 ~GrafoMatriz()

```
GrafoMatriz::~GrafoMatriz ()  [virtual]
```

Destruidor da classe GrafoMatriz. Libera a memória alocada para as matrizes e vetores.

4.4.3 Member Function Documentation

4.4.3.1 calcularIndiceLinear()

```
int GrafoMatriz::calcularIndiceLinear (
            int origem,
            int destino)
```

Calcula o índice linear na matriz comprimida para grafos não direcionados.

Parameters

origem	O id do vértice de origem.
destino	O id do vértice de destino.

Returns

O índice linear correspondente aos vértices de origem e destino.

4.4.3.2 get_aresta()

```
int GrafoMatriz::get_aresta (
            int origem,
            int destino)  [override], [virtual]
```

Método para obter o peso de uma aresta entre dois vértices dados seus ids.

Parameters

origem	O id do vértice de origem.
destino	O id do vértice de destino.

Returns

O peso da aresta entre os vértices de origem e destino.

Implements Grafo.

4.4.3.3 get_vertice()

```
int GrafoMatriz::get_vertice (
            int vertice)  [override], [virtual]
```

Método para obter o peso de um vértice dado seu id.

Parameters

vertice	O id do vértice.

Returns

O peso do vértice correspondente ao id.

Implements Grafo.

4.4.3.4 get_vizinhos()

```
int GrafoMatriz::get_vizinhos (
            int vertice)  [override], [virtual]
```

Método para obter o número de vizinhos de um vértice.

Parameters

vertice	O id do vértice.

Returns

O número de vizinhos do vértice.

Implements Grafo.

4.4.3.5 get_vizinhos_array()

```
int * GrafoMatriz::get_vizinhos_array (
            int id,
            int & tamanho)  [override], [virtual]
```

Implements Grafo.

4.4.3.6 get_vizinhos_vertices()

```
int * GrafoMatriz::get_vizinhos_vertices (
            int vertice,
            int & quantidadeVizinhos)  [override], [virtual]
```

Obtém os vértices vizinhos de um determinado vértice.

Esta função retorna um array contendo os vértices vizinhos do vértice especificado. A quantidade de vizinhos é armazenada na variável passada por referência.

Parameters

vertice	O vértice cujo vizinhos serão buscados.
quantidadeVizinhos	Referência para armazenar a quantidade total de vizinhos.

Returns

Ponteiro para um array contendo os vértices vizinhos.

Implements Grafo.

4.4.3.7 inicializa_grafo()

```
void GrafoMatriz::inicializa_grafo ()  [virtual]
```

Método para inicializar o grafo a partir de um arquivo de entrada. Lê os vértices, arestas e pesos do arquivo de dados.

Implements Grafo.

4.4.3.8 nova_aresta()

```
void GrafoMatriz::nova_aresta (
            int origem,
            int destino,
            int peso)  [virtual]
```

Método para adicionar uma nova aresta entre dois vértices, verificando se a aresta já existe.

Parameters

origem	O id do vértice de origem.
destino	O id do vértice de destino.
peso	O peso da nova aresta.

Implements Grafo.

4.4.3.9 redimensionarMatriz()

```
void GrafoMatriz::redimensionarMatriz ()
```

Método para redimensionar a matriz 2D de adjacência.

4.4.3.10 redimensionarMatrizLinear()

```
void GrafoMatriz::redimensionarMatrizLinear ()
```

Método para redimensionar a matriz linear de adjacência.

4.4.3.11 set_aresta()

```
void GrafoMatriz::set_aresta (
            int origem,
            int destino,
            float peso)  [override], [virtual]
```

Método para definir o peso de uma aresta entre dois vértices dados seus ids.

Parameters

origem	O id do vértice de origem.
destino	O id do vértice de destino.
peso	O peso a ser atribuído à aresta.

Implements Grafo.

4.4.3.12 set_vertice()

```
void GrafoMatriz::set_vertice (
            int id,
            float peso)  [override], [virtual]
```

Método para definir o peso de um vértice dado seu id.

Parameters

id	O id do vértice.
peso	O peso a ser atribuído ao vértice.

Implements Grafo.

The documentation for this class was generated from the following files:

- include/GrafoMatriz.h
- src/GrafoMatriz.cpp

4.5 ListaEncadeada< T > Class Template Reference

A classe ListaEncadeada é uma implementação genérica de uma lista encadeada, capaz de armazenar elementos de qualquer tipo. Esta classe fornece métodos para adicionar, remover e imprimir elementos da lista, além de acessar o primeiro e o último elemento.

```
#include <ListaEncadeada.h>
```

Public Member Functions

- ListaEncadeada ()
  Construtor padrão da lista encadeada. Inicializa a lista com primeiro e último ponteiro nulos.
- T * getInicio () const
  Obtém o primeiro nó da lista.
- void adicionar (T *novoNo)
  Adiciona um novo nó ao final da lista.
- void imprimir () const
  Imprime todos os elementos da lista.
- void remover (T *noParaRemover)
  Remove um nó específico da lista.
- int get_tamanho ()
  Retorna o tamanho atual da lista (número de elementos armazenados).
- ~ListaEncadeada ()
  Destruidor da lista encadeada. Libera toda a memória alocada para os nós da lista.

4.5.1 Detailed Description

template<typename T>
class ListaEncadeada< T >

A classe ListaEncadeada é uma implementação genérica de uma lista encadeada, capaz de armazenar elementos de qualquer tipo. Esta classe fornece métodos para adicionar, remover e imprimir elementos da lista, além de acessar o primeiro e o último elemento.

Template Parameters

T	Tipo dos elementos armazenados na lista.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 ListaEncadeada()

```
template<typename T>
ListaEncadeada< T >::ListaEncadeada ()  [inline]
```

Construtor padrão da lista encadeada. Inicializa a lista com primeiro e último ponteiro nulos.

4.5.2.2 ~ListaEncadeada()

```
template<typename T>
ListaEncadeada< T >::~ListaEncadeada ()  [inline]
```

Destruidor da lista encadeada. Libera toda a memória alocada para os nós da lista.

4.5.3 Member Function Documentation

4.5.3.1 adicionar()

```
template<typename T>
void ListaEncadeada< T >::adicionar (
            T * novoNo)  [inline]
```

Adiciona um novo nó ao final da lista.

Parameters

novoNo	Ponteiro para o nó a ser adicionado.

4.5.3.2 get_tamanho()

```
template<typename T>
int ListaEncadeada< T >::get_tamanho ()  [inline]
```

Retorna o tamanho atual da lista (número de elementos armazenados).

Returns

O número de elementos na lista.

4.5.3.3 getInicio()

```
template<typename T>
T * ListaEncadeada< T >::getInicio () const  [inline]
```

Obtém o primeiro nó da lista.

Returns

O ponteiro para o primeiro nó.

4.5.3.4 imprimir()

```
template<typename T>
void ListaEncadeada< T >::imprimir () const  [inline]
```

Imprime todos os elementos da lista.

Note

A impressão é feita chamando o operador << do tipo `T` para cada elemento da lista.

4.5.3.5 remover()

```
template<typename T>
void ListaEncadeada< T >::remover (
            T * noParaRemover)  [inline]
```

Remove um nó específico da lista.

Parameters

noParaRemover	Ponteiro para o nó a ser removido.

Note

Se o nó não for encontrado ou a lista estiver vazia, a operação não será realizada.

The documentation for this class was generated from the following file:

- include/ListaEncadeada.h

4.6 VerticeEncadeado Class Reference

A classe VerticeEncadeado representa um vértice em um grafo implementado usando uma lista encadeada. Ela armazena informações sobre o vértice, como seu identificador, peso e grau, e as conexões (arestas) com outros vértices.

`#include <VerticeEncadeado.h>`

Public Member Functions

- VerticeEncadeado (int id, int peso)
  Construtor que cria um vértice com um identificador e peso especificados.
- int getId () const
  Obtém o identificador do vértice.
- int getPeso () const
  Obtém o peso do vértice.
- int getGrau () const
  Obtém o grau do vértice, que é o número de conexões (arestas) que ele possui.
- VerticeEncadeado * getProximo () const
  Obtém o próximo vértice na lista encadeada.
- void setProximo (VerticeEncadeado *novoProximo)
  Define o próximo vértice na lista encadeada.
- void setConexao (VerticeEncadeado *verticeDestino, int pesoAresta)
  Estabelece uma conexão entre este vértice e outro vértice, criando uma aresta.
- ArestaEncadeada * getPrimeiraConexao ()
  Obtém a primeira conexão (aresta) do vértice.
- ListaEncadeada< ArestaEncadeada > * getConexoes ()
  Obtém a lista de todas as conexões (arestas) do vértice.
- void setConexoes (ListaEncadeada< ArestaEncadeada > *novasConexoes)
  Define as conexões (arestas) do vértice.
- int removeConexao (VerticeEncadeado *destino)
  Remove uma conexão (aresta) entre este vértice e outro vértice de destino.
- ArestaEncadeada * getConexao (int origem, int destino)
  Obtém uma conexão (aresta) específica entre dois vértices.

Friends

- std::ostream & operator<< (std::ostream &os, const VerticeEncadeado &vertice)
  Sobrecarga do operador de saída, permitindo a impressão do vértice no formato desejado.

4.6.1 Detailed Description

A classe VerticeEncadeado representa um vértice em um grafo implementado usando uma lista encadeada. Ela armazena informações sobre o vértice, como seu identificador, peso e grau, e as conexões (arestas) com outros vértices.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 VerticeEncadeado()

```
VerticeEncadeado::VerticeEncadeado (
            int id,
            int peso)
```

Construtor que cria um vértice com um identificador e peso especificados.

Parameters

id	Identificador do vértice.
peso	Peso do vértice.

4.6.3 Member Function Documentation

4.6.3.1 getConexao()

```
ArestaEncadeada * VerticeEncadeado::getConexao (
            int origem,
            int destino)
```

Obtém uma conexão (aresta) específica entre dois vértices.

Parameters

origem	O vértice de origem da aresta.
destino	O vértice de destino da aresta.

Returns

A aresta encontrada entre os dois vértices.

4.6.3.2 getConexoes()

```
ListaEncadeada< ArestaEncadeada > * VerticeEncadeado::getConexoes ()
```

Obtém a lista de todas as conexões (arestas) do vértice.

Returns

Ponteiro para a lista de arestas.

4.6.3.3 getGrau()

```
int VerticeEncadeado::getGrau () const
```

Obtém o grau do vértice, que é o número de conexões (arestas) que ele possui.

Returns

O grau do vértice.

4.6.3.4 getId()

```
int VerticeEncadeado::getId () const
```

Obtém o identificador do vértice.

Returns

O identificador do vértice.

4.6.3.5 getPeso()

```
int VerticeEncadeado::getPeso () const
```

Obtém o peso do vértice.

Returns

O peso do vértice.

4.6.3.6 getPrimeiraConexao()

```
ArestaEncadeada * VerticeEncadeado::getPrimeiraConexao ()
```

Obtém a primeira conexão (aresta) do vértice.

Returns

Ponteiro para a primeira aresta conectada a este vértice.

4.6.3.7 getProximo()

```
VerticeEncadeado * VerticeEncadeado::getProximo () const
```

Obtém o próximo vértice na lista encadeada.

Returns

O próximo vértice.

4.6.3.8 removeConexao()

```
int VerticeEncadeado::removeConexao (
            VerticeEncadeado * destino)
```

Remove uma conexão (aresta) entre este vértice e outro vértice de destino.

Parameters

destino	O vértice de destino da aresta a ser removida.

Returns

O peso da aresta removida.

4.6.3.9 setConexao()

```
void VerticeEncadeado::setConexao (
            VerticeEncadeado * verticeDestino,
            int pesoAresta)
```

Estabelece uma conexão entre este vértice e outro vértice, criando uma aresta.

Parameters

verticeDestino	O vértice de destino da aresta.
pesoAresta	O peso da aresta.

4.6.3.10 setConexoes()

```
void VerticeEncadeado::setConexoes (
            ListaEncadeada< ArestaEncadeada > * novasConexoes)
```

Define as conexões (arestas) do vértice.

Parameters

novasConexoes	Ponteiro para a nova lista de arestas.

4.6.3.11 setProximo()

```
void VerticeEncadeado::setProximo (
            VerticeEncadeado * novoProximo)
```

Define o próximo vértice na lista encadeada.

Parameters

novoProximo	Ponteiro para o próximo vértice.

4.6.4 Friends And Related Symbol Documentation

4.6.4.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
            const VerticeEncadeado & vertice)  [friend]
```

Sobrecarga do operador de saída, permitindo a impressão do vértice no formato desejado.

Parameters

os	Fluxo de saída.
vertice	O vértice a ser impresso.

Returns

O fluxo de saída.

The documentation for this class was generated from the following files:

- include/VerticeEncadeado.h
- src/VerticeEncadeado.cpp

Chapter 5

File Documentation

5.1 include/ArestaEncadeada.h File Reference

```
#include <iostream>
```

Classes

- class ArestaEncadeada

5.2 ArestaEncadeada.h

Go to the documentation of this file.

```
00001 #ifndef ARESTAENCADEADA_H_INCLUDED
00002 #define ARESTAENCADEADA_H_INCLUDED
00003
00004 #include <iostream>
00005
00006 // A classe ArestaEncadeada representa uma aresta em um grafo, que conecta dois vértices (origem e
      destino)
00007 // e pode ter um peso associado. Esta classe é usada para representar arestas encadeadas em um grafo
      baseado
00008 // em listas encadeadas.
00009 class VerticeEncadeado;  // Declaração antecipada da classe VerticeEncadeado (classe de vértices no
      grafo)
00010
00011 class ArestaEncadeada {
00012 private:
00013     VerticeEncadeado* origem;   // Ponteiro para o vértice de origem da aresta
00014     VerticeEncadeado* destino;  // Ponteiro para o vértice de destino da aresta
00015     float peso;                 // Peso da aresta (pode ser 0 para arestas não ponderadas)
00016     ArestaEncadeada* proximo;  // Ponteiro para a próxima aresta, caso haja uma lista de arestas
      encadeadas
00017
00018 public:
00019     // Construtor que inicializa os valores da aresta (origem, destino e peso).
00020     // O próximo ponteiro é inicializado como nullptr (sem aresta subsequente).
00021     ArestaEncadeada(VerticeEncadeado* origem, VerticeEncadeado* destino, float peso);
00022
00023     // Métodos de acesso (getters) para os membros privados da classe.
00024
00025     // Retorna o vértice de origem da aresta.
00026     VerticeEncadeado* getOrigem() const;
00027
00028     // Retorna o vértice de destino da aresta.
00029     VerticeEncadeado* getDestino() const;
00030
00031     // Retorna o peso da aresta.
```

```
00032     float getPeso() const;
00033
00034     // Retorna o ponteiro para a próxima aresta na lista de arestas encadeadas.
00035     ArestaEncadeada* getProximo() const;
00036
00037     // Define o ponteiro da próxima aresta na lista encadeada de arestas.
00038     void setProximo(ArestaEncadeada* novoProximo);
00039
00040     // Sobrecarga do operador de fluxo («) para permitir a impressão das arestas
00041     // A impressão inclui as informações de origem, destino e peso da aresta.
00042     friend std::ostream& operator«(std::ostream& os, const ArestaEncadeada& aresta);
00043 };
00044
00045 #endif // ARESTAENCADEADA_H_INCLUDED
```

5.3 include/Grafo.h File Reference

```
#include <iostream>
#include <fstream>
#include <ctime>
```

Classes

- class Grafo

  Classe base para a representação de um grafo.

5.4 Grafo.h

Go to the documentation of this file.

```
00001 #ifndef GRAFO_H_INCLUDED
00002 #define GRAFO_H_INCLUDED
00003 #include <iostream>
00004 #include <fstream>
00005 #include <ctime>
00006
00007 using namespace std;
00008
00016 class Grafo
00017 {
00018 private:
00019     bool direcionado;
00020     bool vtp;
00021     bool atp;
00022     int ordem;
00023     int origem;
00024     int destino;
00025     int peso;
00026     int *clusters;    // Array para armazenar os clusters de cada vértice
00027
00028     float modulo_subtracao(int a, int b)
00029     {
00030         if (a > b)
00031         {
00032             return a - b;
00033         }
00034         else if (a < b)
00035         {
00036             return b - a;
00037         }
00038         else
00039         {
00040             return 1;
00041         }
00042     }
00043
00044     const int MAX_NODES = 50000;
00045     const int MAX_EDGES = 1000000;
```

```
00046
00047     struct NodeMapping
00048     {
00049         int original;
00050         int mapped;
00051     };
00052
00053     int mapeia_vertice(NodeMapping node_mapping[], int &node_count, int node, int &next_id)
00054     {
00055         for (int i = 0; i < node_count; i++)
00056         {
00057             if (node_mapping[i].original == node)
00058             {
00059                 return node_mapping[i].mapped; // Retorna o ID já mapeado
00060             }
00061         }
00062         node_mapping[node_count].original = node;
00063         node_mapping[node_count].mapped = next_id;
00064         set_vertice(next_id, 1);
00065         node_count++;
00066         return next_id++; // Retorna o novo ID e incrementa o próximo ID
00067     }
00068
00069 public:
00073     Grafo() = default;
00074
00080     void carrega_grafo_novo(int d)
00081     {
00082         // int d = 1; // Aqui você pode mudar para testar diferentes arquivos
00083         std::string filename;
00084
00085         if (d == 1)
00086         {
00087             filename = "./entradas/aa4.mtx";
00088         } else if (d == 2) {
00089             filename = "./entradas/bio-grid-fruitfly.mtx";
00090         } else if (d == 3) {
00091             filename = "./entradas/bio-grid-yeast.mtx";
00092         } else if (d == 4) {
00093             filename = "./entradas/ca-Erdos992.mtx";
00094         } else if (d == 5) {
00095             filename = "./entradas/airfoil1_dual.mtx";
00096         } else if (d == 6) {
00097             filename = "./entradas/EX5.mtx";
00098         } else if (d == 7) {
00099             filename = "./entradas/ukerbe1.mtx";
00100         } else if (d == 8) {
00101             filename = "./entradas/as-735.mtx";
00102         } else if (d == 9) {
00103             filename = "./entradas/p2p-Gnutella08.mtx";
00104         } else if (d == 10) {
00105             filename = "./entradas/bio-dmela.mtx";
00106         } else {
00107             std::cout « "Arquivo Inválido" « std::endl;
00108         }
00109         // Abre o arquivo para leitura
00110         std::ifstream file(filename);
00111         if (!file.is_open())
00112         {
00113             std::cerr « "Erro ao abrir o arquivo " « filename « std::endl;
00114             return;
00115         }
00116
00117         set_eh_direcionado(false);
00118         set_aresta_ponderada(true);
00119         set_vertice_ponderado(false);
00120
00121         NodeMapping node_mapping[MAX_NODES];
00122         int node_count = 0; // Contador de nós mapeados
00123         int edge_count = 0; // Contador de arestas
00124         int next_id = 1;    // Próximo ID sequencial a ser atribuído
00125
00126         int node1, node2;
00127
00128         // Lê o arquivo linha por linha
00129         while (file » node1 » node2)
00130         {
00131
00132             int mapped_node1 = mapeia_vertice(node_mapping, node_count, node1, next_id);
00133             int mapped_node2 = mapeia_vertice(node_mapping, node_count, node2, next_id);
00134             if (node1 != node2)
00135             {
00136                 set_aresta(mapped_node1, mapped_node2, modulo_subtracao(mapped_node1, mapped_node2));
00137                 edge_count++;
00138             }
00139         }
00140
```

```
00141         file.close();
00142
00143         set_ordem(node_count);
00144
00145         // Cria clusters aleatórios após carregar o grafo
00146         criar_clusters_aleatorios(10); // Exemplo: 3 clusters
00147     };
00148
00152     // virtual ~Grafo() = default;
00153     virtual ~Grafo()
00154     {
00155         if (clusters)
00156         {
00157             delete[] clusters; // Libera a memória alocada para os clusters
00158         }
00159     }
00160
00168     virtual int get_aresta(int origem, int destino) = 0;
00169
00176     virtual int get_vertice(int vertice) = 0;
00177
00184     virtual int get_vizinhos(int vertice) = 0;
00185
00193     virtual void nova_aresta(int origem, int destino, int peso) = 0;
00194
00202     virtual void set_aresta(int origem, int destino, float peso) = 0;
00203
00210     virtual void set_vertice(int id, float peso) = 0;
00211
00223     virtual int *get_vizinhos_vertices(int vertice, int &qtdvizinhos) = 0;
00224
00225     virtual int *get_vizinhos_array(int id, int &tamanho) = 0;
00226
00232     int get_ordem()
00233     {
00234         return ordem;
00235     };
00236
00242     void set_ordem(int ordem)
00243     {
00244         this->ordem = ordem;
00245     };
00246
00250     void aumenta_ordem()
00251     {
00252         this->ordem++;
00253     };
00254
00260     bool eh_direcionado()
00261     {
00262         return direcionado;
00263     }
00264
00270     void set_eh_direcionado(bool direcionado)
00271     {
00272         this->direcionado = direcionado;
00273     };
00274
00280     bool vertice_ponderado()
00281     {
00282         return vtp;
00283     }
00284
00290     void set_vertice_ponderado(bool verticePonderado)
00291     {
00292         this->vtp = verticePonderado;
00293     };
00294
00300     bool aresta_ponderada()
00301     {
00302         return atp;
00303     }
00304
00310     void set_aresta_ponderada(bool arestaPonderada)
00311     {
00312         this->atp = arestaPonderada;
00313     };
00314
00320     int get_grau()
00321     {
00322         if (!eh_direcionado())
00323         {
00324             int grauMaximo = 0;
00325             for (int i = 1; i <= 100; i++)
00326             {
00327                 // int quantidadeVizinhos = get_vizinhos(i);
00328                 // int* vizinhos = get_vizinhos_vertices(i, quantidadeVizinhos);
```

```
00329                 // if (vizinhos) {
00330                 //     cout « "Vizinhos do vértice " « i « ": ";
00331                 //     for (int i = 0; i < quantidadeVizinhos; i++) {
00332                 //         cout « vizinhos[i] « " ";
00333                 //     }
00334                 //     cout « endl;
00335 
00336                 //     // Libera a memória alocada para o array de vizinhos
00337                 //     delete[] vizinhos;
00338                 // } else {
00339                 //     cout « "Vértice inválido ou sem vizinhos." « endl;
00340                 // }
00341                 int numVizinhos = get_vizinhos(i);
00342 
00343                 if (numVizinhos > grauMaximo)
00344                 {
00345                     grauMaximo = numVizinhos;
00346                 }
00347             }
00348             return grauMaximo;
00349         }
00350         else
00351         {
00352             int maxGrauSaida = 0;
00353 
00354             for (int i = 1; i <= ordem; i++)
00355             {
00356                 int grauSaida = 0;
00357 
00358                 // Calcula grau de saída
00359                 int numVizinhos = get_vizinhos(i);
00360                 grauSaida = numVizinhos;
00361 
00362                 if (grauSaida > maxGrauSaida)
00363                 {
00364                     maxGrauSaida = grauSaida;
00365                 }
00366             }
00367             return maxGrauSaida;
00368         }
00369     }
00370 
00376     bool eh_completo()
00377     {
00378         for (int i = 1; i <= get_ordem(); i++)
00379         {
00380             if (get_vizinhos(i) < get_ordem() - 1)
00381                 return false;
00382         }
00383         return true;
00384     }
00385 
00392     void dfs(int vertice, bool visitado[])
00393     {
00394         visitado[vertice] = true;
00395         for (int i = 1; i <= ordem; i++)
00396         {
00397             if (get_aresta(vertice, i) && !visitado[i])
00398             {
00399                 dfs(i, visitado);
00400             }
00401         }
00402     }
00403 
00409     int n_conexo()
00410     {
00411 
00412         bool *visitado = new bool[ordem + 1]; // Usa alocação dinâmica para evitar problemas de
      tamanho
00413         for (int i = 1; i <= ordem; i++)
00414         {                         // Se os vértices começam em 1
00415             visitado[i] = false; // Inicializa corretamente
00416         }
00417 
00418         int componentes = 0;
00419 
00420         for (int i = 1; i <= ordem; i++)
00421         { // Se os vértices começam em 1
00422             if (!visitado[i])
00423             {                       // Usa índice corretamente
00424                 dfs(i, visitado); // Chama a DFS
00425                 componentes++;
00426             }
00427         }
00428 
00429         delete[] visitado; // Libera memória alocada dinamicamente
00430         return componentes;
```

```
00431     }
00432
00436     virtual void inicializa_grafo() = 0;
00437
00445     void maior_menor_distancia()
00446     {
00447         int n = get_ordem();
00448
00449         if (n == 0)
00450         {
00451             cout « "O grafo está vazio." « endl;
00452             return;
00453         }
00454
00455         // Matriz de distâncias
00456         int dist[n + 1][n + 1];
00457
00458         // Inicializa a matriz de distâncias
00459         for (int i = 1; i <= n; i++)
00460         {
00461             for (int j = 1; j <= n; j++)
00462             {
00463                 if (i == j)
00464                 {
00465                     dist[i][j] = 0; // Distância de um nó para ele mesmo
00466                 }
00467                 else
00468                 {
00469                     int peso = get_aresta(i, j);
00470                     dist[i][j] = (peso > 0) ? peso : 999999; // Se não há aresta, assume um valor alto
    (infinito)
00471                 }
00472             }
00473         }
00474
00475         // Algoritmo de Floyd-Warshall
00476         for (int k = 1; k <= n; k++)
00477         {
00478             for (int i = 1; i <= n; i++)
00479             {
00480                 for (int j = 1; j <= n; j++)
00481                 {
00482                     if (dist[i][k] != 999999 && dist[k][j] != 999999)
00483                     {
00484                         if (dist[i][j] > dist[i][k] + dist[k][j])
00485                         {
00486                             dist[i][j] = dist[i][k] + dist[k][j];
00487                         }
00488                     }
00489                 }
00490             }
00491         }
00492
00493         // Encontrar os nós mais distantes
00494         int maxDist = -1;
00495         int no1 = -1, no2 = -1;
00496
00497         for (int i = 1; i <= n; i++)
00498         {
00499             for (int j = i + 1; j <= n; j++)
00500             {
00501                 if (dist[i][j] != 999999 && dist[i][j] > maxDist)
00502                 {
00503                     maxDist = dist[i][j];
00504                     no1 = i;
00505                     no2 = j;
00506                 }
00507             }
00508         }
00509
00510         // Exibir resultado
00511         if (no1 != -1 && no2 != -1)
00512         {
00513             cout « "Maior menor distância: (" « no1 « "-" « no2 « ") " « maxDist « endl;
00514         }
00515         else
00516         {
00517             cout « "Não há caminho entre os nós." « endl;
00518         }
00519     }
00520
00521     void criar_clusters_aleatorios(int num_clusters)
00522     {
00523         if (clusters)
00524         {
00525             delete[] clusters; // Libera a memória anterior, se houver
00526         }
```

```
00527         clusters = new int[ordem + 1]; // Aloca memória para os clusters
00528
00529         for (int i = 1; i <= ordem; i++)
00530         {
00531             // Obtém o último dígito do número do vértice
00532             int ultimo_digito = i % 10;
00533
00534             // Atribui o cluster com base no último dígito
00535
00536             clusters[i] = i % 71 == 0 ? (ultimo_digito % num_clusters) + 1 : 0;
00537         }
00538     }
00539
00540     int *get_clusters()
00541     {
00542         return clusters;
00543     }
00544
00552     float encontrar_agmg_guloso()
00553     {
00554         int num_clusters = 10;
00555
00556         bool *vertices_visitados = new bool[ordem + 1]();        // Inicializa todos como false
00557         bool *cluster_conectados = new bool[num_clusters + 1](); // Inicializa todos como false
00558         float soma_pesos_agmg = 0;
00559
00560         int vertice_atual = 1;                                   // Começa no vértice 1
00561         vertices_visitados[vertice_atual] = true;                // Marca o vértice inicial como visitado
00562         cluster_conectados[clusters[vertice_atual]] = true; // Marca o cluster do vértice inicial como
      conectado
00563
00564         while (true)
00565         {
00566             std::cout << "soma_pesos_agmg: " << soma_pesos_agmg << std::endl;
00567
00568             int qtd_vizinhos;
00569             int *vizinhos = get_vizinhos_vertices(vertice_atual, qtd_vizinhos);
00570
00571             if (qtd_vizinhos == 0)
00572             {
00573                 delete[] vizinhos;
00574                 std::cout << "Nao foi possivel encontrar uma arvore que ligue todos os clusters " <<
      std::endl;
00575                 break;
00576             }
00577
00578             float menor_peso_local = 99999;
00579             int proximo_vertice = -1;
00580
00581             for (int i = 0; i < qtd_vizinhos; i++)
00582             {
00583                 int vizinho = vizinhos[i];
00584                 float peso_aresta = get_aresta(vertice_atual, vizinho);
00585
00586                 if (!cluster_conectados[clusters[vizinho]])
00587                 {
00588                     if (peso_aresta < menor_peso_local)
00589                     {
00590                         menor_peso_local = peso_aresta;
00591                         proximo_vertice = vizinho;
00592                     }
00593
00594                     else if (peso_aresta == menor_peso_local)
00595                     {
00596                     }
00597                 }
00598             }
00599
00600             if (proximo_vertice == -1)
00601             {
00602                 for (int i = 0; i < qtd_vizinhos; i++)
00603                 {
00604                     int vizinho = vizinhos[i];
00605                     float peso_aresta = get_aresta(vertice_atual, vizinho);
00606
00607                     if (!vertices_visitados[vizinho] && peso_aresta < menor_peso_local)
00608                     {
00609                         menor_peso_local = peso_aresta;
00610                         proximo_vertice = vizinho;
00611                     }
00612                 }
00613             }
00614
00615             if (proximo_vertice == -1)
00616             {
00617                 delete[] vizinhos;
00618                 std::cout << "Nao foi possivel encontrar uma arvore que ligue todos os clusters " <<
```

```
    std::endl;
00619                 break;
00620             }
00621
00622             vertice_atual = proximo_vertice;
00623             vertices_visitados[vertice_atual] = true;
00624             cluster_conectados[clusters[vertice_atual]] = true;
00625             soma_pesos_agmg += menor_peso_local;
00626
00627             bool todos_clusters_conectados = true;
00628             for (int i = 1; i <= num_clusters; i++)
00629             {
00630                 if (!cluster_conectados[i])
00631                 {
00632
00633                     todos_clusters_conectados = false;
00634                     break;
00635                 }
00636             }
00637
00638             delete[] vizinhos;
00639
00640             if (todos_clusters_conectados)
00641             {
00642                 break;
00643             }
00644         }
00645
00646         delete[] cluster_conectados;
00647         delete[] vertices_visitados;
00648
00649         return soma_pesos_agmg;
00650     }
00651
00659     float encontrar_agmg_randomizado()
00660     {
00661         int num_clusters = 10;
00662
00663         bool *vertices_visitados = new bool[ordem + 1]();
00664         bool *cluster_conectados = new bool[num_clusters + 1]();
00665         float soma_pesos_agmg = 0;
00666
00667         std::srand(std::time(0));
00668
00669         int vertice_atual = std::rand() % ordem + 1;
00670         vertices_visitados[vertice_atual] = true;
00671         cluster_conectados[clusters[vertice_atual]] = true;
00672
00673         while (true)
00674         {
00675             std::cout << "vertice_atual: " << vertice_atual << std::endl;
00676
00677             int qtd_vizinhos;
00678             int *vizinhos = get_vizinhos_vertices(vertice_atual, qtd_vizinhos);
00679
00680             if (qtd_vizinhos == 0)
00681             {
00682                 delete[] vizinhos;
00683                 std::cout << "Nao foi possivel encontrar uma arvore que ligue todos os clusters " <<
    std::endl;
00684                 break;
00685             }
00686
00687             float menor_peso_local = 99999;
00688             int candidatos[ordem];
00689             int num_candidatos = 0;
00690
00691             for (int i = 0; i < qtd_vizinhos; i++)
00692             {
00693                 int vizinho = vizinhos[i];
00694                 float peso_aresta = get_aresta(vertice_atual, vizinho);
00695
00696                 if (!vertices_visitados[vizinho] && peso_aresta <= menor_peso_local)
00697                 {
00698                     if (peso_aresta < menor_peso_local)
00699                     {
00700                         menor_peso_local = peso_aresta;
00701                         num_candidatos = 0;
00702                     }
00703                     candidatos[num_candidatos++] = vizinho;
00704                 }
00705
00706                 if (!cluster_conectados[clusters[vizinho]])
00707                 {
00708                     std::cout << "cluster do " << vizinho << " e " << clusters[vizinho] << std::endl;
00709                     num_candidatos = 0;
00710                     candidatos[num_candidatos++] = vizinho;
```

```
00711                     break;
00712                 }
00713             }
00714
00715             if (num_candidatos == 0)
00716             {
00717                 delete[] vizinhos;
00718                 std::cout « "Nao foi possivel encontrar uma arvore que ligue todos os clusters " «
      std::endl;
00719                 break;
00720             }
00721
00722             int proximo_vertice = candidatos[std::rand() % num_candidatos];
00723
00724             vertice_atual = proximo_vertice;
00725             vertices_visitados[vertice_atual] = true;
00726             cluster_conectados[clusters[vertice_atual]] = true;
00727             soma_pesos_agmg += menor_peso_local;
00728
00729             bool todos_clusters_conectados = true;
00730             for (int i = 1; i <= num_clusters; i++)
00731             {
00732                 if (!cluster_conectados[i])
00733                 {
00734                     todos_clusters_conectados = false;
00735                     break;
00736                 }
00737             }
00738
00739             delete[] vizinhos;
00740
00741             if (todos_clusters_conectados)
00742             {
00743                 break;
00744             }
00745         }
00746
00747         delete[] cluster_conectados;
00748         delete[] vertices_visitados;
00749
00750         return soma_pesos_agmg;
00751     }
00752
00753     void testa_get_vizinhos(int id)
00754     {
00755         int *vizinhos = new int[ordem];
00756         int qtd_vizinhos;
00757         vizinhos = get_vizinhos_vertices(id, qtd_vizinhos);
00758
00759         cout « "vizinhos do " « id « " ";
00760         for (int i = 0; i < qtd_vizinhos; i++)
00761         {
00762             cout « vizinhos[i] « " ";
00763         }
00764         cout « endl;
00765     }
00766 };
00767
00768 #endif // GRAFO_H_INCLUDED
```

5.5 include/GrafoLista.h File Reference

```
#include "Grafo.h"
#include "ListaEncadeada.h"
#include "VerticeEncadeado.h"
#include "ArestaEncadeada.h"
#include <iostream>
```

Classes

- class GrafoLista

  A classe GrafoLista é uma implementação de grafo que utiliza listas encadeadas para armazenar os vértices e arestas. Ela herda da classe abstrata Grafo e implementa suas funções virtuais para manipulação de vértices e arestas.

5.6 GrafoLista.h

Go to the documentation of this file.

```
00001 #ifndef GRAFOLISTA_H_INCLUDED
00002 #define GRAFOLISTA_H_INCLUDED
00003
00004 #include "Grafo.h"
00005 #include "ListaEncadeada.h"
00006 #include "VerticeEncadeado.h"
00007 #include "ArestaEncadeada.h"
00008
00009 #include <iostream>
00010
00011 using namespace std;
00012
00017 class GrafoLista : public Grafo
00018 {
00019 private:
00023     ListaEncadeada<VerticeEncadeado> *vertices;
00024
00028     ListaEncadeada<ArestaEncadeada> *arestas;
00029
00036     VerticeEncadeado *get_vertice_encadeado(int id);
00037
00044     void buscaEmProfundidade(VerticeEncadeado *vertice, bool *visitados);
00045
00046 public:
00050     GrafoLista();
00051
00058     int get_vertice(int id) override;
00059
00067     int get_aresta(int idOrigem, int idDestino) override;
00068
00075     void set_vertice(int id, float peso) override;
00076
00084     void set_aresta(int origem, int destino, float peso) override;
00085
00094     void nova_aresta(int origem, int destino, int peso) override;
00095
00102     int get_vizinhos(int vertice) override;
00103
00107     void imprimir();
00108
00113     void inicializa_grafo() override;
00114
00115     int* get_vizinhos_array(int id, int& tamanho) override;
00116
00128     int* get_vizinhos_vertices(int vertice, int& quantidadeVizinhos) override;
00129
00133     ~GrafoLista();
00134
00135
00136 };
00137
00138 #endif // GRAFOLISTA_H_INCLUDED
```

5.7 include/GrafoMatriz.h File Reference

```
#include "Grafo.h"
#include <string>
```

Classes

- class GrafoMatriz

  A classe GrafoMatriz implementa a interface da classe abstrata Grafo utilizando uma matriz de adjacência. A classe gerencia tanto grafos direcionados quanto não direcionados, além de permitir a manipulação de pesos de vértices e arestas.

Variables

- const int TAMANHO_INICIAL = 10

5.7.1 Variable Documentation

5.7.1.1 TAMANHO_INICIAL

```
const int TAMANHO_INICIAL = 10
```

5.8 GrafoMatriz.h

Go to the documentation of this file.

```
00001 #ifndef GRAFO_MATRIZ_H_INCLUDED
00002 #define GRAFO_MATRIZ_H_INCLUDED
00003
00004 #include "Grafo.h"
00005 #include <string>
00006
00007 using namespace std;
00008
00009 const int TAMANHO_INICIAL = 10; // Começa com 10 vértices
00010
00015 class GrafoMatriz : public Grafo {
00016 private:
00020     int** Matriz;
00021
00025     int* MatrizLinear;
00026
00030     int* VetorPesosVertices;
00031
00035     int tamanhoAtual;
00036
00040     int tamanhoAtualLinear;
00041
00042 public:
00046     GrafoMatriz();
00047
00051     virtual ~GrafoMatriz();
00052
00056     void redimensionarMatriz();
00057
00061     void redimensionarMatrizLinear();
00062
00067     void inicializa_grafo();
00068
00076     int calcularIndiceLinear(int origem, int destino);
00077
00085     int get_aresta(int origem, int destino) override;
00086
00093     int get_vertice(int vertice) override;
00094
00101     int get_vizinhos(int vertice) override;
00102
00109     void set_vertice(int id, float peso) override;
00110
00118     void set_aresta(int origem, int destino, float peso) override;
00119
00127     void nova_aresta(int origem, int destino, int peso);
00128
00129     int* get_vizinhos_array(int id, int& tamanho) override;
00130
00142     int* get_vizinhos_vertices(int vertice, int& quantidadeVizinhos) override;
00143 };
00144
00145 #endif  // GRAFO_MATRIZ_H_INCLUDED
```

5.9 include/ListaEncadeada.h File Reference

```
#include <iostream>
```

Classes

- class ListaEncadeada< T >

  A classe ListaEncadeada é uma implementação genérica de uma lista encadeada, capaz de armazenar elementos de qualquer tipo. Esta classe fornece métodos para adicionar, remover e imprimir elementos da lista, além de acessar o primeiro e o último elemento.

5.10 ListaEncadeada.h

Go to the documentation of this file.

```
00001 #ifndef LISTAENCADEADA_H_INCLUDED
00002 #define LISTAENCADEADA_H_INCLUDED
00003
00004 #include <iostream>
00005
00006 using namespace std;
00007
00014 template <typename T>
00015
00016 class ListaEncadeada {
00017 private:
00021     T* primeiro;
00022
00026     T* ultimo;
00027
00031     int tamanho;
00032
00033 public:
00037     ListaEncadeada() : primeiro(nullptr), ultimo(nullptr), tamanho(0) {}
00038
00044     T* getInicio() const {
00045         return primeiro;
00046     }
00047
00053     void adicionar(T* novoNo) {
00054         if (primeiro == nullptr) {
00055             primeiro = novoNo;
00056             ultimo = novoNo;
00057         } else {
00058             ultimo->setProximo(novoNo);
00059             ultimo = novoNo;
00060         }
00061         tamanho++;
00062     }
00063
00069     void imprimir() const {
00070         T* atual = primeiro;
00071         while (atual != nullptr) {
00072             cout << *atual << endl;
00073             atual = atual->getProximo();
00074         }
00075     }
00076
00083     void remover(T* noParaRemover) {
00084         if(!primeiro || !noParaRemover) {
00085             return;
00086         }
00087         if(primeiro == noParaRemover) {
00088             primeiro = primeiro->getProximo();
00089             if (!primeiro) {
00090                 ultimo = nullptr;
00091             }
00092             tamanho--;
00093             delete noParaRemover;
00094             return;
00095         }
00096
00097         T* atual = primeiro;
00098         while(atual->getProximo() && atual->getProximo() != noParaRemover) {
00099             atual = atual->getProximo();
00100         }
00101
00102         if(atual->getProximo() == noParaRemover) {
00103             atual->setProximo(noParaRemover->getProximo());
00104
00105             if (noParaRemover == ultimo) {
00106                 ultimo = atual;
00107             }
```

```
00108             tamanho--;
00109             delete noParaRemover;
00110         }
00111     }
00112
00118     int get_tamanho() {
00119         return tamanho;
00120     }
00121
00125     ~ListaEncadeada() {
00126         T* atual = primeiro;
00127         while (atual != nullptr) {
00128             T* proximo = atual->getProximo();
00129             delete atual;
00130             atual = proximo;
00131         }
00132     }
00133 };
00134
00135 #endif // LISTAENCADEADA_H_INCLUDED
```

5.11 include/VerticeEncadeado.h File Reference

```
#include <iostream>
#include "ListaEncadeada.h"
#include "ArestaEncadeada.h"
```

Classes

- class VerticeEncadeado

  A classe VerticeEncadeado representa um vértice em um grafo implementado usando uma lista encadeada. Ela armazena informações sobre o vértice, como seu identificador, peso e grau, e as conexões (arestas) com outros vértices.

5.12 VerticeEncadeado.h

Go to the documentation of this file.

```
00001 #ifndef VERTICEENCADEADO_H_INCLUDED
00002 #define VERTICEENCADEADO_H_INCLUDED
00003
00004 #include <iostream>
00005 #include "ListaEncadeada.h"
00006 #include "ArestaEncadeada.h"
00007
00012 class VerticeEncadeado {
00013 private:
00017     int id;
00018
00022     int peso;
00023
00027     int grau;
00028
00032     VerticeEncadeado* proximo;
00033
00037     ListaEncadeada<ArestaEncadeada>* conexoes;
00038
00039 public:
00046     VerticeEncadeado(int id, int peso);
00047
00053     int getId() const;
00054
00060     int getPeso() const;
00061
00067     int getGrau() const;
00068
00074     VerticeEncadeado* getProximo() const;
00075
```

```
00081     void setProximo(VerticeEncadeado* novoProximo);
00082
00089     void setConexao(VerticeEncadeado* verticeDestino, int pesoAresta);
00090
00096     ArestaEncadeada* getPrimeiraConexao();
00097
00103     ListaEncadeada<ArestaEncadeada>* getConexoes();
00104
00110     void setConexoes(ListaEncadeada<ArestaEncadeada>* novasConexoes);
00111
00120     friend std::ostream& operator<<(std::ostream& os, const VerticeEncadeado& vertice);
00121
00129     int removeConexao(VerticeEncadeado* destino);
00130
00139     ArestaEncadeada* getConexao(int origem, int destino);
00140 };
00141
00142 #endif // VERTICEENCADEADO_H_INCLUDED
```

5.13 src/ArestaEncadeada.cpp File Reference

```
#include "../include/ArestaEncadeada.h"
#include "../include/VerticeEncadeado.h"
#include <iostream>
```

Functions

- std::ostream & operator<< (std::ostream &os, const ArestaEncadeada &aresta)

5.13.1 Function Documentation

5.13.1.1 operator<<()

```
std::ostream & operator<< (
            std::ostream & os,
            const ArestaEncadeada & aresta)
```

5.14 src/GrafoLista.cpp File Reference

```
#include <iostream>
#include <fstream>
#include "../include/GrafoLista.h"
```

5.15 src/GrafoMatriz.cpp File Reference

```
#include "../include/GrafoMatriz.h"
#include "../include/Grafo.h"
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <ctime>
```

Variables

- const int TAMANHO_FIXO = 10000

5.15.1 Variable Documentation

5.15.1.1 TAMANHO_FIXO

```
const int TAMANHO_FIXO = 10000
```

5.16 src/VerticeEncadeado.cpp File Reference

```
#include "../include/VerticeEncadeado.h"
```

Functions

- std::ostream & operator<< (std::ostream &os, const VerticeEncadeado &vertice)

5.16.1 Function Documentation

5.16.1.1 operator<<()

```
std::ostream & operator<< (
            std::ostream & os,
            const VerticeEncadeado & vertice)
```

Parameters

os	Fluxo de saída.
vertice	O vértice a ser impresso.

Returns

O fluxo de saída.