Trabalho 1 e 2 de Teoria dos Grafos

1.0

Generated by Doxygen 1.13.1

Chapter 1

Trabalho_de_grafos

Aluno: Lukas Freitas de Carvalho - 202376033

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

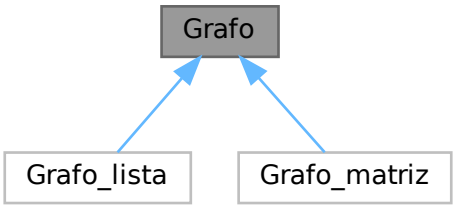
Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 Grafo Class Reference

Inheritance diagram for Grafo:



Public Member Functions

- virtual **NodeVertex** * **getVertice** (int id)=0

  Retorna o nó caso exista do id desejado.
- virtual **NodeEdge** * **getAresta** (int origem, int destino)=0

  Retorna a aresta caso exista.
- virtual void **insereVertice** (float val)=0

  Insere um vértice com peso e o coloca em último.
- virtual void **insereAresta** (int origem, int destino, float val)=0

  Insere uma aresta entre dois nós com peso e o coloca em último caso não exista no mesmo ponto.
- virtual void **removeVertice** (int id)=0

  Remove o vértice com o id-1 passado como parâmetro.
- virtual void **removeAresta** (int origem, int destino)=0

  Remove a aresta caso exista.
- bool **eh_direcionado** ()

  Retorna se o grafo é direcionado ou não.
- bool **verticePonderado** ()

Retorna se os vértices do grafo são ponderados ou não.

- bool **arestaPonderada** ()

  Retorna se as arestas do grafo são ponderadas ou não.
- int **getOrdem** ()

  Retorna a ordem do grafo.
- void **setOrdem** (int val)

  Define a ordem do grafo.
- void **setDirecionado** (bool val)

  Define se o grafo é direcionado ou não.
- void **setVerticePonderado** (bool val)

  Define se o grafo possui vértices ponderados.
- void **setArestaPonderada** (bool val)

  Define se o grafo possui arestas ponderadas.
- int **getGrau** ()

  Retorna o grau do grafo.
- bool **eh_completo** ()

  Retorna se o grafo é completo.
- void **carregaGrafo** (string grafo)

  Carrega o grafo na estrutura.
- void **imprimeGrafo** ()

  Imprime o grafo de acordo com a descrição do trabalho.
- float **maiorMenorDistancia** (int ponto1, int ponto2)

  Calcula a menor distância entre dois pontos utilizando o algoritmo de Dijkstra.

5.1.1 Member Function Documentation

5.1.1.1 arestaPonderada()

```
bool Grafo::arestaPonderada ()
```

Retorna se as arestas do grafo são ponderadas ou não.

Returns

Verdadeiro ou falso

5.1.1.2 carregaGrafo()

```
void Grafo::carregaGrafo (
            string grafo)
```

Carrega o grafo na estrutura.

Parameters

grafo	String que representa o nome do arquivo à ser carregado

5.1.1.3 eh_completo()

```
bool Grafo::eh_completo ()
```

Retorna se o grafo é completo.

Returns

booleano representando se é ou não completo

5.1.1.4 eh_direcionado()

```
bool Grafo::eh_direcionado ()
```

Retorna se o grafo é direcionado ou não.

Returns

Verdadeiro ou falso

5.1.1.5 getAresta()

```
virtual  NodeEdge * Grafo::getAresta (
             int origem,
             int destino)  [pure virtual]
```

Retorna a aresta caso exista.

Parameters

origem	Indica o nó de origem na aresta (origem começa em zero)
destino	Indica o nó de destino na aresta (destino começa em zero)

Returns

Retorna a aresta caso exista

Implemented in **Grafo_lista** (p. 16), and **Grafo_matriz** (p. 19).

5.1.1.6 getGrau()

```
int Grafo::getGrau ()
```

Retorna o grau do grafo.

Returns

Inteiro representado pelo grau

5.1.1.7 getOrdem()

```
int Grafo::getOrdem ()
```

Retorna a ordem do grafo.

Returns

Inteiro representado pela ordem

5.1.1.8 getVertice()

```
virtual NodeVertex * Grafo::getVertice (
            int id) [pure virtual]
```

Retorna o nó caso exista do id desejado.

Parameters

id	ID do nó deslocado uma unidade

Returns

Retorna o nó caso exista

Implemented in **Grafo_lista** (p. 16), and **Grafo_matriz** (p. 19).

5.1.1.9 insereAresta()

```
virtual void Grafo::insereAresta (
            int origem,
            int destino,
            float val) [pure virtual]
```

Insere uma aresta entre dois nós com peso e o coloca em último caso não exista no mesmo ponto.

Parameters

origem	Indica o nó onde a aresta vai ser colocada
destino	Indica o nó ao qual estará ligado
val	Valor a ser inserido na aresta

Implemented in **Grafo_lista** (p. 17), and **Grafo_matriz** (p. 20).

5.1.1.10 insereVertice()

```
virtual void Grafo::insereVertice (
            float val) [pure virtual]
```

Insere um vértice com peso e o coloca em último.

Parameters

val	Valor a ser inserido no vértice

Implemented in **Grafo_lista** (p. 17), and **Grafo_matriz** (p. 20).

5.1.1.11 maiorMenorDistancia()

```
float Grafo::maiorMenorDistancia (
            int ponto1,
            int ponto2)
```

Calcula a menor distância entre dois pontos utilizando o algoritmo de Dijkstra.

Parameters

ponto1	Nó de origem
ponto2	Nó de destino

Returns

Menor distância entre dois pontos

5.1.1.12 removeAresta()

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

Remove a aresta caso exista.

Parameters

origem	Indica o nó de origem (origem começa em 1)
destino	Indica o nó de destino (destino começa em 1)

Implemented in **Grafo_lista** (p. 17), and **Grafo_matriz** (p. 20).

5.1.1.13 removeVertice()

```
virtual void Grafo::removeVertice (
            int id)  [pure virtual]
```

Remove o vértice com o id-1 passado como parâmetro.

Parameters

id	Indica o nó que será removido (O id sempre é uma unidade a mais)

Implemented in **Grafo_lista** (p. 17), and **Grafo_matriz** (p. 20).

5.1.1.14 setArestaPonderada()

```
void Grafo::setArestaPonderada (
            bool val)
```

Define se o grafo possui arestas ponderadas.

Parameters

val	Valor que será definido para as arestas ponderadas

5.1.1.15 setDirecionado()

```
void Grafo::setDirecionado (
            bool val)
```

Define se o grafo é direcionado ou não.

Parameters

val	Valor que será definido para o direcionamento do grafo

5.1.1.16 setOrdem()

```
void Grafo::setOrdem (
            int val)
```

Define a ordem do grafo.

Parameters

val	Valor que será definido para a ordem do grafo

5.1.1.17 setVerticePonderado()

```
void Grafo::setVerticePonderado (
            bool val)
```

Define se o grafo possui vértices ponderados.

Parameters

val	Valor que será definido para os vértices ponderados

5.1.1.18 verticePonderado()

```
bool Grafo::verticePonderado ()
```

Retorna se os vértices do grafo são ponderados ou não.

Returns

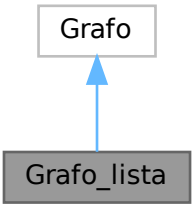
Verdadeiro ou falso

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

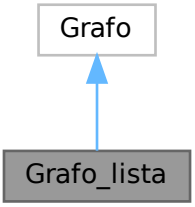
- include/Grafo.h
- src/Grafo.cpp

5.2 Grafo_lista Class Reference

Inheritance diagram for Grafo_lista:



Collaboration diagram for Grafo_lista:



Public Member Functions

- **Grafo_lista** ()

  *Construtor da classe **Grafo** (p. 9) lista.*
- ∼**Grafo_lista** ()

  *Destrutor da classe **Grafo** (p. 9) lista.*
- void **insereVertice** (float val) override
- void **insereAresta** (int origem, int destino, float val) override
- void **removeAresta** (int i, int j) override
- void **removeVertice** (int id) override
- **NodeVertex** ∗ **getVertice** (int id) override
- **NodeEdge** ∗ **getAresta** (int origem, int destino) override

Public Member Functions inherited from Grafo

- bool **eh_direcionado** ()

  Retorna se o grafo é direcionado ou não.
- bool **verticePonderado** ()

  Retorna se os vértices do grafo são ponderados ou não.
- bool **arestaPonderada** ()

  Retorna se as arestas do grafo são ponderadas ou não.
- int **getOrdem** ()

  Retorna a ordem do grafo.
- void **setOrdem** (int val)

  Define a ordem do grafo.
- void **setDirecionado** (bool val)

  Define se o grafo é direcionado ou não.
- void **setVerticePonderado** (bool val)

  Define se o grafo possui vértices ponderados.
- void **setArestaPonderada** (bool val)

  Define se o grafo possui arestas ponderadas.
- int **getGrau** ()

  Retorna o grau do grafo.
- bool **eh_completo** ()

  Retorna se o grafo é completo.
- void **carregaGrafo** (string grafo)

  Carrega o grafo na estrutura.
- void **imprimeGrafo** ()

  Imprime o grafo de acordo com a descrição do trabalho.
- float **maiorMenorDistancia** (int ponto1, int ponto2)

  Calcula a menor distância entre dois pontos utilizando o algoritmo de Dijkstra.

5.2.1 Member Function Documentation

5.2.1.1 getAresta()

```
NodeEdge * Grafo_lista::getAresta (
            int origem,
            int destino)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 11).

5.2.1.2 getVertice()

```
NodeVertex * Grafo_lista::getVertice (
            int id)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 12).

5.2.1.3 insereAresta()

```
void Grafo_lista::insereAresta (
            int origem,
            int destino,
            float val)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 12).

5.2.1.4 insereVertice()

```
void Grafo_lista::insereVertice (
            float val)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 12).

5.2.1.5 removeAresta()

```
void Grafo_lista::removeAresta (
            int i,
            int j)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 13).

5.2.1.6 removeVertice()

```
void Grafo_lista::removeVertice (
            int id)  [override], [virtual]
```

@inheritDoc

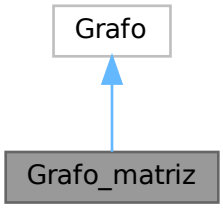
Implements **Grafo** (p. 13).

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

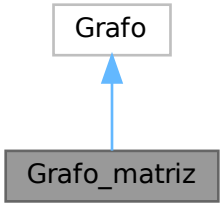
- include/Grafo_lista.h
- src/Grafo_lista.cpp

5.3 Grafo_matriz Class Reference

Inheritance diagram for Grafo_matriz:



Collaboration diagram for Grafo_matriz:



Public Member Functions

- **Grafo_matriz** ()

  Construtor da classe matriz.
- ~**Grafo_matriz** ()

  Destrutor da classe matriz.
- void **insereVertice** (float val) override
- void **insereAresta** (int origem, int destino, float val) override
- void **removeAresta** (int origem, int destino) override
- void **removeVertice** (int id) override
- **NodeVertex** * **getVertice** (int id) override
- **NodeEdge** * **getAresta** (int origem, int destino) override

Public Member Functions inherited from Grafo

- bool **eh_direcionado** ()

  Retorna se o grafo é direcionado ou não.
- bool **verticePonderado** ()

  Retorna se os vértices do grafo são ponderados ou não.
- bool **arestaPonderada** ()

  Retorna se as arestas do grafo são ponderadas ou não.
- int **getOrdem** ()

  Retorna a ordem do grafo.
- void **setOrdem** (int val)

  Define a ordem do grafo.
- void **setDirecionado** (bool val)

  Define se o grafo é direcionado ou não.
- void **setVerticePonderado** (bool val)

  Define se o grafo possui vértices ponderados.
- void **setArestaPonderada** (bool val)

  Define se o grafo possui arestas ponderadas.
- int **getGrau** ()

  Retorna o grau do grafo.
- bool **eh_completo** ()

  Retorna se o grafo é completo.
- void **carregaGrafo** (string grafo)

  Carrega o grafo na estrutura.
- void **imprimeGrafo** ()

  Imprime o grafo de acordo com a descrição do trabalho.
- float **maiorMenorDistancia** (int ponto1, int ponto2)

  Calcula a menor distância entre dois pontos utilizando o algoritmo de Dijkstra.

5.3.1 Member Function Documentation

5.3.1.1 getAresta()

```
NodeEdge * Grafo_matriz::getAresta (
            int origem,
            int destino)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 11).

5.3.1.2 getVertice()

```
NodeVertex * Grafo_matriz::getVertice (
            int id)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 12).

5.3.1.3 insereAresta()

```
void Grafo_matriz::insereAresta (
            int origem,
            int destino,
            float val)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 12).

5.3.1.4 insereVertice()

```
void Grafo_matriz::insereVertice (
            float val)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 12).

5.3.1.5 removeAresta()

```
void Grafo_matriz::removeAresta (
            int origem,
            int destino)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 13).

5.3.1.6 removeVertice()

```
void Grafo_matriz::removeVertice (
            int id)  [override], [virtual]
```

@inheritDoc

Implements **Grafo** (p. 13).

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

- include/Grafo_matriz.h
- src/Grafo_matriz.cpp

5.4 Linked_list< NodeType > Class Template Reference

Classe template para criação do grafo por lista encadeada.

```
#include <Linked_list.hpp>
```

Public Member Functions

- **Linked_list** ()
  Construtor da Lista Encadeada.
- ~**Linked_list** ()
  Destrutor da Lista encadeada.
- int **getTam** ()
  Retorna o tamanho da lista.
- NodeType * **getNodeById** (int val)
  Retorna o nó correspondente ao id passado.
- NodeType * **getUltimo** ()
  Retorna o ultimo nó
- NodeType * **getPrimeiro** ()
  Retorna o primeiro nó
- void **insereFinal** (float val)
  Insere nó no final da lista.
- void **imprimeLista** ()
  Imprime a Lista encadeada.
- void **removeNode** (NodeType *no)
  Remove o nó passado como parâmetro da lista.

Protected Member Functions

- void **limpaNodes** ()
  Função auxiliar que deleta todos os nós.

Protected Attributes

- NodeType * **primeiro**
- NodeType * **ultimo**

5.4.1 Detailed Description

template<typename NodeType>
class Linked_list< NodeType >

Classe template para criação do grafo por lista encadeada.

Template Parameters

NodeType	Tipo de nó que será utilizado

5.4.2 Member Function Documentation

5.4.2.1 getNodeById()

```
template<typename NodeType>
NodeType *  Linked_list< NodeType >::getNodeById (
             int val)
```

Retorna o nó correspondente ao id passado.

Parameters

val	Representa o id do nó a ser passado

5.4.2.2 getPrimeiro()

```
template<typename NodeType>
NodeType * Linked_list< NodeType >::getPrimeiro ()
```

Retorna o primeiro nó

Returns

Primeiro nó

5.4.2.3 getTam()

```
template<typename NodeType>
int Linked_list< NodeType >::getTam ()
```

Retorna o tamanho da lista.

Returns

Tamanho da lista

5.4.2.4 getUltimo()

```
template<typename NodeType>
NodeType * Linked_list< NodeType >::getUltimo ()
```

Retorna o ultimo nó

Returns

Último nó

5.4.2.5 insereFinal()

```
template<typename NodeType>
void Linked_list< NodeType >::insereFinal (
            float val)
```

Insere nó no final da lista.

Parameters

val	Valor a ser inserido no nó

5.4.2.6 removeNode()

```
template<typename NodeType>
void  Linked_list< NodeType >::removeNode (
            NodeType * no)
```

Remove o nó passado como parâmetro da lista.

Parameters

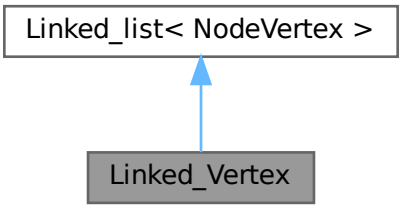
no	Nó a ser removido

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

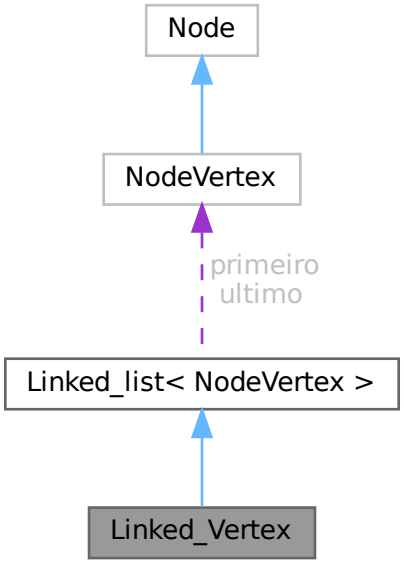
- include/Linked_list.hpp

5.5 Linked_Vertex Class Reference

Inheritance diagram for Linked_Vertex:



Collaboration diagram for Linked_Vertex:



Public Member Functions

- **Linked_Vertex** ()

  Construtor da classe Lista encadeada para os vértices.
- ∼**Linked_Vertex** ()

  Destrutor da classe lista encadeada para os vértices.
- void **insereAresta** (int origem, int destino, float val)

  Insere uma aresta entre dois vértices.
- void **removeAresta** (int i, int j)

  Remove uma aresta entre dois vértices.
- void **removeVertice** (int id)

  Remove um vértice.

Public Member Functions inherited from Linked_list< NodeVertex >

- **Linked_list** ()

  Construtor da Lista Encadeada.
- ∼**Linked_list** ()

  Destrutor da Lista encadeada.
- int **getTam** ()

  Retorna o tamanho da lista.
- **NodeVertex** ∗ **getNodeById** (int val)

  Retorna o nó correspondente ao id passado.
- **NodeVertex** ∗ **getUltimo** ()

  Retorna o ultimo nó

- **NodeVertex** ∗ **getPrimeiro** ()

  Retorna o primeiro nó
- void **insereFinal** (float val)

  Insere nó no final da lista.
- void **imprimeLista** ()

  Imprime a Lista encadeada.
- void **removeNode** (**NodeVertex** ∗no)

  Remove o nó passado como parâmetro da lista.

Additional Inherited Members

Protected Member Functions inherited from Linked_list< NodeVertex >

- void **limpaNodes** ()

  Função auxiliar que deleta todos os nós.

Protected Attributes inherited from Linked_list< NodeVertex >

- **NodeVertex** ∗ **primeiro**
- **NodeVertex** ∗ **ultimo**

5.5.1 Member Function Documentation

5.5.1.1 insereAresta()

```
void Linked_Vertex::insereAresta (
            int origem,
            int destino,
            float val)
```

Insere uma aresta entre dois vértices.

Parameters

origem	Define o nó em que será inserido a aresta
destino	Define o id do nó que a aresta corresponde
val	Define o peso da aresta

5.5.1.2 removeAresta()

```
void Linked_Vertex::removeAresta (
            int i,
            int j)
```

Remove uma aresta entre dois vértices.

Parameters

i	Nó em que será removida a aresta
j	Id do nó que a aresta corresponde

5.5.1.3 removeVertice()

```
void Linked_Vertex::removeVertice (
            int id)
```

Remove um vértice.

Parameters

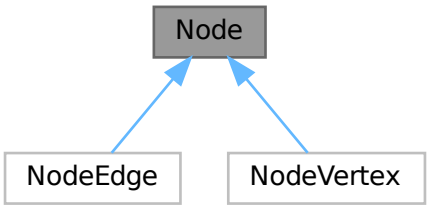
id	Nó que será removido

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

- include/Linked_Vertex.h
- src/Linked_Vertex.cpp

5.6 Node Class Reference

Inheritance diagram for Node:



Public Member Functions

- **Node** ()

  Construtor da classe.
- ∼**Node** ()

  Destrutor da classe.
- **Node** ∗ **getProx** ()

  Retorna o próximo nó
- void **setProx** (**Node** ∗prox)

Define o próximo nó

- void **setValue** (float value)

  Define o valor do nó
- float **getValue** ()

  Retorna o valor do nó
- void **setId** (int val)

  Define o id do nó
- int **getId** ()

  Retorna o id do nó

5.6.1 Member Function Documentation

5.6.1.1 getId()

```
int Node::getId ()
```

Retorna o id do nó

Returns

ID do nó

5.6.1.2 getProx()

```
Node * Node::getProx ()
```

Retorna o próximo nó

Returns

Próximo nó

5.6.1.3 getValue()

```
float Node::getValue ()
```

Retorna o valor do nó

Returns

Valor do nó

5.6.1.4 setId()

```
void Node::setId (
            int val)
```

Define o id do nó

Parameters

val	Id do nó

5.6.1.5 setProx()

```
void Node::setProx (
            Node * prox)
```

Define o próximo nó

Parameters

prox	Nó que será inserido como próximo nó

5.6.1.6 setValue()

```
void Node::setValue (
            float value)
```

Define o valor do nó

Parameters

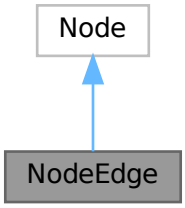
value	valor do nó

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

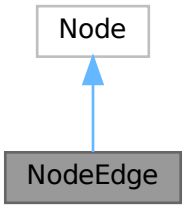
- include/Node.h
- src/Node.cpp

5.7 NodeEdge Class Reference

Inheritance diagram for NodeEdge:



Collaboration diagram for NodeEdge:



Public Member Functions

- **NodeEdge** ()
- ∼**NodeEdge** ()
- float **getPeso** ()
  Retorna o peso da aresta.
- void **setPeso** (float val)
  Define o peso da aresta.

Public Member Functions inherited from Node

- **Node** ()
  Construtor da classe.
- ∼**Node** ()
  Destrutor da classe.
- **Node** ∗ **getProx** ()
  Retorna o próximo nó
- void **setProx** (**Node** ∗prox)
  Define o próximo nó
- void **setValue** (float value)
  Define o valor do nó
- float **getValue** ()
  Retorna o valor do nó
- void **setId** (int val)
  Define o id do nó
- int **getId** ()
  Retorna o id do nó

5.7.1 Constructor & Destructor Documentation

5.7.1.1 NodeEdge()

```
NodeEdge::NodeEdge ()
```

@inheritDoc

5.7.1.2 ~NodeEdge()

```
NodeEdge::~NodeEdge ()
```

@inheritDoc

5.7.2 Member Function Documentation

5.7.2.1 getPeso()

```
float NodeEdge::getPeso ()
```

Retorna o peso da aresta.

Returns

Peso

5.7.2.2 setPeso()

```
void NodeEdge::setPeso (
            float val)
```

Define o peso da aresta.

Parameters

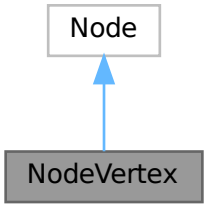
val	Peso da aresta

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

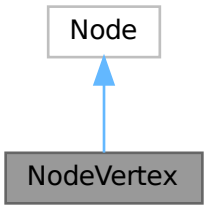
- include/NodeEdge.h
- src/NodeEdge.cpp

5.8 NodeVertex Class Reference

Inheritance diagram for NodeVertex:



Collaboration diagram for NodeVertex:



Public Member Functions

- **NodeVertex** ()
- ~**NodeVertex** ()
- **Linked_list**< **NodeEdge** > * **getArestas** ()
  Retorna a lista de arestas do vértice.
- int **getGrau** ()
  Retorna o tamanho da lista de arestas do vértice (seu grau)
- void **setGrau** (int val)
  Define o grau do vértice.

Public Member Functions inherited from Node

- **Node** ()
  Construtor da classe.
- ~**Node** ()
  Destrutor da classe.
- **Node** * **getProx** ()
  Retorna o próximo nó
- void **setProx** (**Node** *prox)
  Define o próximo nó
- void **setValue** (float value)
  Define o valor do nó
- float **getValue** ()
  Retorna o valor do nó
- void **setId** (int val)
  Define o id do nó
- int **getId** ()
  Retorna o id do nó

5.8.1 Constructor & Destructor Documentation

5.8.1.1 NodeVertex()

```
NodeVertex::NodeVertex ()
```

@inheritDoc

5.8.1.2 ~NodeVertex()

```
NodeVertex::~NodeVertex ()
```

@inheritDoc

5.8.2 Member Function Documentation

5.8.2.1 getArestas()

```
Linked_list< NodeEdge > * NodeVertex::getArestas ()
```

Retorna a lista de arestas do vértice.

Returns

Lista encadeada de arestas

5.8.2.2 getGrau()

```
int NodeVertex::getGrau ()
```

Retorna o tamanho da lista de arestas do vértice (seu grau)

Returns

Grau do vértice

5.8.2.3 setGrau()

```
void NodeVertex::setGrau (
            int val)
```

Define o grau do vértice.

Parameters

val	Novo grau do vértice

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

- include/NodeVertex.h
- src/NodeVertex.cpp

Chapter 6

File Documentation

6.1 Grafo.h

```
00001 #ifndef GRAFO_H
00002 #define GRAFO_H
00003 #include "NodeVertex.h"
00004 #include "NodeEdge.h"
00005
00006
00007 class Grafo
00008 {
00009     private:
00010         int ordem = 0;
00011         bool direcionado, verticePeso, arestaPeso;
00012
00018         string imprmeSimNao(bool valor);
00019
00024         string retornaMaiorMenorDistancia();
00025
00026     public:
00032     virtual NodeVertex* getVertice(int id) = 0;
00033
00040     virtual NodeEdge* getAresta(int origem, int destino) = 0;
00041
00046     virtual void insereVertice(float val) = 0;
00047
00054     virtual void insereAresta(int origem, int destino, float val) = 0;
00055
00060     virtual void removeVertice(int id) = 0;
00061
00067     virtual void removeAresta(int origem, int destino) = 0;
00068
00073     bool eh_direcionado();
00074
00079     bool verticePonderado();
00080
00085     bool arestaPonderada();
00086
00091     int getOrdem();
00092
00097     void setOrdem(int val);
00098
00103     void setDirecionado(bool val);
00104
00109     void setVerticePonderado(bool val);
00110
00115     void setArestaPonderada(bool val);
00116
00121     int getGrau();
00122
00127     bool eh_completo();
00128
00133     void carregaGrafo(string grafo);
00134
00138     void imprimeGrafo();
00139
00146     float maiorMenorDistancia(int ponto1, int ponto2);
00147
00148 };
00149
00150 #include "../src/Grafo.cpp"
00151
00152 #endif
```

6.2 Grafo_lista.h

```
00001 #ifndef GRAFO_LISTA_H
00002 #define GRAFO_LISTA_H
00003
00004 #include "Grafo.h"
00005 #include "Linked_Vertex.h"
00006 #include "NodeEdge.h"
00007 #include "NodeVertex.h"
00008
00009 class Grafo_lista : public Grafo
00010 {
00011     private:
00012         Linked_Vertex* Vertice;
00013     public:
00017         Grafo_lista();
00021         ~Grafo_lista();
00022
00026         void insereVertice(float val) override;
00027
00031         void insereAresta(int origem, int destino, float val) override;
00032
00036         void removeAresta(int i, int j) override;
00037
00041         void removeVertice(int id) override;
00042
00046         NodeVertex* getVertice(int id) override;
00047
00051         NodeEdge* getAresta(int origem, int destino) override;
00052 };
00053
00054 #include "../src/Grafo_lista.cpp"
00055
00056 #endif
```

6.3 Grafo_matriz.h

```
00001 #ifndef GRAFO_MATRIZ_H
00002 #define GRAFO_MATRIZ_H
00003
00004 #include "Grafo.h"
00005 #include "NodeEdge.h"
00006 #include "NodeVertex.h"
00007
00008 class Grafo_matriz : public Grafo
00009 {
00010     private:
00011         NodeEdge*** matriz_adjacencia;
00012         NodeVertex* vertices;
00013         int capacidade;
00014
00018         void inicializaMatriz();
00019
00023         void inicializaPesoVertices();
00024
00031         NodeEdge** retornaCelulaMatriz(int i, int j);
00032
00037         void resize(int novaCapacidade);
00038
00039     public:
00040
00044         Grafo_matriz();
00045
00049         ~Grafo_matriz();
00050
00054         void insereVertice(float val) override;
00055
00059         void insereAresta(int origem, int destino, float val) override;
00060
00064         void removeAresta(int origem, int destino) override;
00065
00069         void removeVertice(int id) override;
00070
00074         NodeVertex* getVertice(int id) override;
00075
00079         NodeEdge* getAresta(int origem, int destino) override;
00080 };
00081
00082 #include "../src/Grafo_matriz.cpp"
00083
00084 #endif
```

6.4 Linked_list.hpp

```
00001 #ifndef LINKED_LIST_HPP
00002 #define LINKED_LIST_HPP
00003
00009 template <typename NodeType>
00010 class Linked_list
00011 {
00012 protected:
00013     NodeType* primeiro, *ultimo;
00014
00018     void limpaNodes();
00019 private:
00020     int n;
00021 public:
00025     Linked_list();
00029     ~Linked_list();
00030
00035     int getTam();
00036
00041     NodeType* getNodeById(int val);
00042
00047     NodeType* getUltimo();
00048
00053     NodeType* getPrimeiro();
00054
00059     void insereFinal(float val);
00060
00064     void imprimeLista();
00065
00070     void removeNode(NodeType* no);
00071 };
00072
00073 #include "Linked_list.tpp"
00074
00075 #endif
```

6.5 Linked_Vertex.h

```
00001     #ifndef LINKED_VERTEX_H
00002     #define LINKED_VERTEX_H
00003
00004     #include "Linked_list.hpp"
00005
00006     class Linked_Vertex : public Linked_list<NodeVertex>
00007     {
00008         public:
00012         Linked_Vertex();
00013
00017         ~Linked_Vertex();
00018
00025         void insereAresta(int origem, int destino, float val);
00026
00032         void removeAresta(int i, int j);
00033
00038         void removeVertice(int id);
00039     };
00040
00041 #include "../src/Linked_Vertex.cpp"
00042
00043 #endif
```

6.6 Node.h

```
00001 #ifndef NODE_H
00002 #define NODE_H
00003
00004 class Node
00005 {
00006 private:
00007     float value;
00008     Node* prox;
00009     int id;
00010
00011 public:
00015     Node();
00016
```

```
00020     ~Node();
00021
00026     Node* getProx();
00027
00032     void setProx(Node* prox);
00033
00038     void setValue(float value);
00039
00044     float getValue();
00045
00050     void setId(int val);
00051
00056     int getId();
00057 };
00058
00059 #include "../src/Node.cpp"
00060
00061 #endif
```

6.7 NodeEdge.h

```
00001 #ifndef NODEEDGE_H
00002 #define NODEEDGE_H
00003 #include "Node.h"
00004
00005 class NodeEdge : public Node
00006 {
00007
00008     private:
00009     float peso;
00010
00011     public:
00015     NodeEdge();
00016
00020     ~NodeEdge();
00021
00026     float getPeso();
00027
00032     void setPeso(float val);
00033
00034 };
00035
00036 #include "../src/NodeEdge.cpp"
00037
00038 #endif
```

6.8 NodeVertex.h

```
00001 #ifndef NODEVERTEX_H
00002 #define NODEVERTEX_H
00003
00004 #include "Node.h"
00005 #include "Linked_list.hpp"
00006 #include "NodeEdge.h"
00007
00008 class NodeVertex : public Node
00009 {
00010
00011     private:
00012     Linked_list<NodeEdge>* Arestas;
00013     int grau;
00014
00015     public:
00019     NodeVertex();
00020
00024     ~NodeVertex();
00025
00030     Linked_list<NodeEdge>* getArestas();
00031
00036     int getGrau();
00037
00042     void setGrau(int val);
00043 };
00044
00045 #include "../src/NodeVertex.cpp"
00046
00047 #endif
```

6.9 Grafo.cpp

```
00001     #include "../include/Grafo.h"
00002     #include <fstream>
00003     #include <iostream>
00004     #include <iomanip>
00005     using namespace std;
00006
00007     bool Grafo::eh_direcionado() {
00008         return this->direcionado;
00009     }
00010     bool Grafo::verticePonderado() {
00011         return this->verticePeso;
00012     }
00013     bool Grafo::arestaPonderada() {
00014         return this->arestaPeso;
00015     }
00016     int Grafo::getOrdem()
00017     {
00018         return this->ordem;
00019     }
00020
00021     void Grafo::setOrdem(int val) {
00022         this->ordem = val;
00023     }
00024
00025     void Grafo::setDirecionado(bool val) {
00026         this->direcionado = val;
00027     }
00028
00029     void Grafo::setVerticePonderado(bool val) {
00030         this->verticePeso = val;
00031     }
00032
00033     void Grafo::setArestaPonderada(bool val) {
00034         this->arestaPeso = val;
00035     }
00036
00037     int Grafo::getGrau(){
00038         NodeVertex* no = getVertice(0);
00039         int maior = no->getGrau();
00040         for(int i = 1; i<getOrdem(); i+=1)
00041         {
00042             no = getVertice(i);
00043             if(no != nullptr && no->getGrau() > maior)
00044             {
00045                 maior = no->getGrau();
00046             }
00047         }
00048         return maior;
00049     };
00050
00051     bool Grafo::eh_completo(){
00052         for(int i = 0; i<getOrdem(); i+=1)
00053         {
00054             NodeVertex* no = getVertice(i);
00055             if(no->getGrau() != getOrdem()-1)
00056             {
00057                 return false;
00058             }
00059         }
00060         return true;
00061     }
00062
00063     void Grafo::carregaGrafo(string grafo) {
00064         string caminhoGrafo = "entradas/" + grafo;
00065         ifstream inFile(caminhoGrafo);
00066
00067         int numVertices, direcionado, arestaPonderada;
00068         float verticePonderado;
00069         inFile » numVertices » direcionado » verticePonderado » arestaPonderada;
00070
00071         this->setDirecionado(direcionado);
00072         this->setVerticePonderado(verticePonderado);
00073         this->setArestaPonderada(arestaPonderada);
00074
00075         if (verticePonderado) {
00076             for (int i = 0; i < numVertices; i+=1) {
00077                 float peso;
00078                 inFile » peso;
00079                 insereVertice(peso);
00080
00081             }
00082         }
00083         else
00084         {
```

```
00085             for (int i = 0; i < numVertices; i+=1) {
00086                 insereVertice(1);
00087             }
00088         }
00089 
00090         int origem, destino;
00091         while (inFile » origem » destino) {
00092             if (arestaPonderada) {
00093                 float peso;
00094                 inFile » peso;
00095                 if(eh_direcionado())
00096                 {
00097                     insereAresta(origem, destino, peso);
00098                 }
00099                 else
00100                 {
00101                     insereAresta(destino, origem, peso);
00102                 }
00103             }
00104             else {
00105                 if(eh_direcionado())
00106                 {
00107                     insereAresta(origem, destino, 1);
00108                 }
00109                 else
00110                 {
00111                     insereAresta(destino, origem, 1);
00112                 }
00113             }
00114         }
00115         inFile.close();
00116     }
00117 
00118     void Grafo::imprimeGrafo()
00119     {
00120         cout«"grafo.txt"«endl;
00121         cout«endl;
00122         cout«"Grau: "«getGrau()«endl;
00123         cout«"Ordem: "«getOrdem()«endl;
00124         cout«"Direcionado: "«imprmeSimNao(eh_direcionado())«endl;
00125         cout«"Vertices ponderados: "«imprmeSimNao(verticePonderado())«endl;
00126         cout«"Arestas ponderadas: "«imprmeSimNao(arestaPonderada())«endl;
00127         cout«"Completo: "«imprmeSimNao(eh_completo())«endl;
00128         cout«"Maior menor distância: "«retornaMaiorMenorDistancia()«endl;
00129     }
00130 
00131     string Grafo::imprmeSimNao(bool valor)
00132     {
00133         if(valor)
00134         {
00135             return "Sim";
00136         }
00137         return "Não";
00138     }
00139 
00140     float Grafo::maiorMenorDistancia(int ponto1, int ponto2) {
00141     NodeVertex* noPontoUm = getVertice(ponto1);
00142     NodeVertex* noPontoDois = getVertice(ponto2);
00143 
00144     if (noPontoUm == nullptr || noPontoDois == nullptr) {
00145         cout « "Erro: Vértices não encontrados." « endl;
00146         return -1;
00147     }
00148     if(noPontoUm->getGrau() == 0 && noPontoDois->getGrau() == 0)
00149     {
00150         return -1;
00151     }
00152 
00153     //Variável grande que eu usei para simular o infinito
00154     const float INF = 1e9;
00155     float* distancias = new float[getOrdem()]();
00156     bool* visitados = new bool[getOrdem()]();
00157 
00158 
00159     for (int i = 0; i < getOrdem(); i+=1) {
00160         distancias[i] = INF;
00161         visitados[i] = false;
00162     }
00163 
00164     distancias[ponto1] = 0;
00165 
00166     for (int count = 0; count < getOrdem(); count+=1) {
00167         int u = -1;
00168         float min_dist = INF;
00169 
00170         for (int i = 0; i < getOrdem(); i+=1) {
00171             if (!visitados[i] && distancias[i] < min_dist) {
```

```
00172                 min_dist = distancias[i];
00173                 u = i;
00174             }
00175         }
00176 
00177         if (u == -1) break;
00178         visitados[u] = true;
00179 
00180         for (int v = 0; v < getOrdem(); v+=1) {
00181             if (u == v) continue;
00182             NodeEdge* aresta = getAresta(u, v);
00183             if (aresta != nullptr) {
00184                 float peso = aresta->getPeso();
00185                 if (distancias[u] + peso < distancias[v]) {
00186                     distancias[v] = distancias[u] + peso;
00187                 }
00188             }
00189         }
00190     }
00191 
00192     float valor = distancias[ponto2];
00193     delete[] distancias;
00194     delete[] visitados;
00195 
00196     if (valor == INF) {
00197         return -1;
00198     } else {
00199         return valor;
00200     }
00201 
00202 }
00203 
00204 string Grafo::retornaMaiorMenorDistancia()
00205 {
00206     if (arestaPonderada()) {
00207         for (int i = 0; i < getOrdem(); i+=1) {
00208             for (int j = 0; j < getOrdem(); j+=1) {
00209                 NodeEdge* aresta = getAresta(i, j);
00210                 if (aresta != nullptr)
00211                 {
00212                     if(aresta->getPeso() < 0){
00213                         return "\nNão é permitido o uso de pesos negativos nas arestas";
00214                     }
00215                 }
00216             }
00217         }
00218     }
00219     float maior = -1;
00220     int indexX = -1;
00221     int indexY = -1;
00222     for(int i = 0; i<getOrdem(); i+=1)
00223     {
00224         for(int j = 0; j<getOrdem(); j+=1)
00225         {
00226             if(i != j)
00227             {
00228                 float valor = maiorMenorDistancia(i,j);
00229                 if(valor > maior)
00230                 {
00231                     indexX = i;
00232                     indexY = j;
00233                     maior = valor;
00234                 }
00235             }
00236         }
00237     }
00238     if(maior == -1)
00239     {
00240         return "\nNão existe nenhum caminho de nenhum vértice para nenhum vértice";
00241     }
00242     else
00243     {
00244         return "(" + to_string(indexX+1) + "-" + to_string(indexY+1) + ") " + to_string(maior);
00245     }
00246 }
```

6.10 Grafo_lista.cpp

```
00001 #include <iostream>
00002 #include "../include/Grafo_lista.h"
00003 
00004 using namespace std;
00005 
```

```
00006 Grafo_lista::Grafo_lista(){
00007     this->Vertice = new Linked_Vertex();
00008 }
00009
00010 Grafo_lista::~Grafo_lista(){
00011     delete Vertice;
00012 }
00013
00014
00015 void Grafo_lista::insereVertice(float val)
00016 {
00017     this->Vertice->insereFinal(val);
00018     setOrdem(getOrdem()+1);
00019 }
00020
00021
00022 void Grafo_lista::insereAresta(int origem, int destino, float val)
00023 {
00024     if(origem >=1 && origem <= getOrdem() && destino >=1 && destino <= getOrdem())
00025     {
00026         if(getAresta(origem-1,destino-1) == nullptr)
00027         {
00028             this->Vertice->insereAresta(origem, destino, val);
00029             NodeVertex* no = this->Vertice->getNodeById(origem-1);
00030             no->setGrau(no->getGrau()+1);
00031             if(!eh_direcionado())
00032             {
00033                 this->Vertice->insereAresta(destino, origem, val);
00034                 NodeVertex* noVolta = this->Vertice->getNodeById(destino-1);
00035                 noVolta->setGrau(noVolta->getGrau()+1);
00036             }
00037         }
00038         else
00039         {
00040             cout«"Aresta entre: "«origem«" e "«destino«" já existe"«endl;
00041         }
00042     }
00043     else
00044     {
00045         cout«"Aresta inválida!"«endl;
00046     }
00047 }
00048
00049 NodeEdge* Grafo_lista::getAresta(int origem, int destino)
00050 {
00051     if(origem >=0 && origem <getOrdem() && destino>=0 && destino < getOrdem())
00052     {
00053         NodeEdge* no = this->Vertice->getNodeById(origem)->getArestas()->getPrimeiro();
00054         if(no != nullptr)
00055         {
00056             while(no!= nullptr && no->getValue() != destino)
00057             {
00058                 no = (NodeEdge*)no->getProx();
00059             }
00060             return no;
00061         }
00062     }
00063     return nullptr;
00064 }
00065
00066 NodeVertex* Grafo_lista::getVertice(int id)
00067 {
00068     return this->Vertice->getNodeById(id);
00069 }
00070
00071 void Grafo_lista::removeAresta(int i, int j)
00072 {
00073     if(i >= 1 && i <=getOrdem() && j>=1 && j <=getOrdem())
00074     {
00075         if(getAresta(i-1,j-1) != nullptr)
00076         {
00077             this->Vertice->removeAresta(i, j);
00078             NodeVertex* no = this->Vertice->getNodeById(i-1);
00079             no->setGrau(no->getGrau()-1);
00080             if(!eh_direcionado())
00081             {
00082                 this->Vertice->removeAresta(j, i);
00083                 NodeVertex* noVolta = this->Vertice->getNodeById(j-1);
00084                 noVolta->setGrau(noVolta->getGrau()-1);
00085             }
00086         }
00087         else
00088         {
00089             cout«"Aresta inexistente!"«endl;
00090         }
00091     }
00092     else
```

```
00093     {
00094         cout«"Não é possível remover o nó"«endl;
00095     }
00096 }
00097
00098 void Grafo_lista::removeVertice(int id)
00099 {
00100
00101
00102     if(id >= 1 && id <= getOrdem())
00103     {
00104         id-=1;
00105         if(id >= 0 && id < getOrdem())
00106         {
00107             this->Vertice->removeVertice(id);
00108             setOrdem(getOrdem()-1);
00109         }
00110     }
00111     else
00112     {
00113         cout«"Não é possível remover o vértice"«endl;
00114     }
00115 }
```

6.11 Grafo_matriz.cpp

```
00001 #include<iostream>
00002 #include "../include/Grafo_matriz.h"
00003 using namespace std;
00004
00005 Grafo_matriz::Grafo_matriz(){
00006     matriz_adjacencia = nullptr;
00007     vertices = nullptr;
00008     capacidade = 10;
00009 }
00010 Grafo_matriz::~Grafo_matriz() {
00011     if (matriz_adjacencia != nullptr) {
00012         if (eh_direcionado()) {
00013             for (int i = 0; i < capacidade; i+=1) {
00014                 for (int j = 0; j < capacidade; j+=1) {
00015                     delete matriz_adjacencia[i][j];
00016                 }
00017                 delete[] matriz_adjacencia[i];
00018             }
00019         } else {
00020             if (matriz_adjacencia[0] != nullptr) {
00021                 int tamanho = capacidade * (capacidade - 1) / 2;
00022                 for (int i = 0; i < tamanho; i+=1) {
00023                     delete matriz_adjacencia[0][i];
00024                 }
00025                 delete[] matriz_adjacencia[0];
00026             }
00027         }
00028         delete[] matriz_adjacencia;
00029     }
00030     delete[] vertices;
00031 }
00032
00033 void Grafo_matriz::inicializaPesoVertices() {
00034     delete[] vertices;
00035     vertices = new NodeVertex[capacidade]();
00036 }
00037 void Grafo_matriz::inicializaMatriz() {
00038     if (eh_direcionado()) {
00039         matriz_adjacencia = new NodeEdge**[capacidade];
00040         for (int i = 0; i < capacidade; i+=1) {
00041             matriz_adjacencia[i] = new NodeEdge*[capacidade]();
00042         }
00043     } else {
00044         int tamanho = capacidade * (capacidade - 1) / 2;
00045         matriz_adjacencia = new NodeEdge**[1];
00046         matriz_adjacencia[0] = new NodeEdge*[tamanho]();
00047     }
00048 }
00049
00050 void Grafo_matriz::resize(int novaCapacidade) {
00051 NodeVertex* newVertices = new NodeVertex[novaCapacidade]();
00052 for (int i = 0; i < getOrdem(); i+=1) {
00053     newVertices[i].setValue(vertices[i].getValue());
00054     newVertices[i].setGrau(vertices[i].getGrau());
00055 }
00056 delete[] vertices;
00057 vertices = newVertices;
```

```
00058
00059 NodeEdge*** novaMatriz = nullptr;
00060 if (eh_direcionado()) {
00061
00062     novaMatriz = new NodeEdge**[novaCapacidade];
00063     for (int i = 0; i < novaCapacidade; i+=1) {
00064         novaMatriz[i] = new NodeEdge*[novaCapacidade]();
00065     }
00066
00067     for (int i = 0; i < capacidade; i+=1) {
00068         for (int j = 0; j < capacidade; j+=1) {
00069             novaMatriz[i][j] = matriz_adjacencia[i][j];
00070         }
00071     }
00072
00073     for (int i = 0; i < capacidade; i+=1) {
00074         delete[] matriz_adjacencia[i];
00075     }
00076 } else {
00077     int novoTamanho = novaCapacidade * (novaCapacidade - 1) / 2;
00078     int tamanhoAntigo = capacidade * (capacidade - 1) / 2;
00079     novaMatriz = new NodeEdge**[1];
00080     novaMatriz[0] = new NodeEdge*[novoTamanho]();
00081
00082     for (int i = 0; i < tamanhoAntigo; i+=1) {
00083         novaMatriz[0][i] = matriz_adjacencia[0][i];
00084     }
00085
00086     delete[] matriz_adjacencia[0];
00087 }
00088
00089 delete[] matriz_adjacencia;
00090 matriz_adjacencia = novaMatriz;
00091 capacidade = novaCapacidade;
00092 }
00093
00094
00095 void Grafo_matriz::insereVertice(float val) {
00096     if (getOrdem() >= capacidade) {
00097         resize(capacidade * 2);
00098     }
00099     if (vertices == nullptr) {
00100         inicializaPesoVertices();
00101     }
00102     vertices[getOrdem()].setValue(val);
00103     setOrdem(getOrdem()+1);
00104 }
00105
00106
00107 void Grafo_matriz::insereAresta(int origem, int destino, float val) {
00108     if(origem < 1 || origem > getOrdem() || destino < 1 || destino > getOrdem()) {
00109         cout<<"Aresta inválida!"<<endl;
00110         return;
00111     }
00112     if(origem != destino)
00113     {
00114         origem -=1;
00115         destino-=1;
00116         if(matriz_adjacencia == nullptr)
00117         {
00118             inicializaMatriz();
00119         }
00120         if(origem >=0 && origem < getOrdem() && destino >=0 && destino < getOrdem())
00121         {
00122             if(getAresta(origem, destino) != nullptr)
00123             {
00124                 cout<<"Aresta inválida!"<<endl;
00125                 return;
00126             }
00127             else
00128             {
00129                 NodeEdge** aresta = retornaCelulaMatriz(origem, destino);
00130                 *aresta = new NodeEdge();
00131                 (*aresta)->setPeso(val);
00132                 (*aresta)->setValue(destino+1);
00133                 vertices[origem].setGrau(vertices[origem].getGrau()+1);
00134                 if(!eh_direcionado())
00135                 {
00136                     vertices[destino].setGrau(vertices[destino].getGrau()+1);
00137                 }
00138             }
00139         }
00140     }
00141 }
00142
00143 NodeEdge** Grafo_matriz::retornaCelulaMatriz(int i, int j)
00144 {
```

```
00145     if(eh_direcionado())
00146     {
00147         return &matriz_adjacencia[i][j];
00148     }
00149     else
00150     {
00151         if(i<j)
00152         {
00153             return &matriz_adjacencia[0][j*(j-1)/2 + i];
00154         }
00155         else
00156         {
00157             return &matriz_adjacencia[0][i*(i-1)/2 + j];
00158         }
00159     }
00160 }
00161
00162 NodeVertex* Grafo_matriz::getVertice(int id)
00163 {
00164     if(id >= getOrdem() || id < 0)
00165     {
00166         return nullptr;
00167     }
00168     return &vertices[id];
00169 }
00170
00171 NodeEdge* Grafo_matriz::getAresta(int origem, int destino)
00172 {
00173     if(origem >=0 && origem < getOrdem() && destino >=0 && destino < getOrdem() && origem != destino)
00174     {
00175         return *retornaCelulaMatriz(origem, destino);
00176     }
00177     return nullptr;
00178 }
00179
00180 void Grafo_matriz::removeAresta(int i, int j)
00181 {
00182     if(i>= 1 && i <=getOrdem() && j>=1 && j<=getOrdem())
00183     {
00184         NodeEdge** arestaPtr = retornaCelulaMatriz(i-1, j-1);
00185
00186         vertices[i-1].setGrau(vertices[i-1].getGrau()-1);
00187         if (*arestaPtr != nullptr) {
00188             cout«"Removendo a aresta ("« i«","«j«")"«endl;
00189             delete *arestaPtr;
00190             *arestaPtr = nullptr;
00191         }
00192         else
00193         {
00194             cout«"Aresta inválida!"«endl;
00195         }
00196     }
00197     else
00198     {
00199         cout«"Aresta inválida!"«endl;
00200     }
00201 }
00202
00203 void Grafo_matriz::removeVertice(int id) {
00204     if (id < 1 || id > getOrdem()) {
00205         cout « "ID inválido!" « endl;
00206         return;
00207     }
00208     int k = id - 1;
00209
00210     for (int i = 0; i < getOrdem(); i+=1) {
00211         if (eh_direcionado()) {
00212             NodeEdge** arestaEntrada = retornaCelulaMatriz(i, k);
00213             if (*arestaEntrada != nullptr) {
00214                 vertices[i].setGrau(vertices[i].getGrau() - 1);
00215             }
00216         }
00217         else {
00218             NodeEdge** aresta = retornaCelulaMatriz(k, i);
00219                 if (*aresta != nullptr) {
00220                     vertices[i].setGrau(vertices[i].getGrau() - 1);
00221                 }
00222         }
00223 }
00224
00225
00226     for(int i = 0; i< getOrdem(); i+=1)
00227         {
00228             NodeEdge** arestaPtr = retornaCelulaMatriz(k, i);
00229             if (*arestaPtr != nullptr) {
00230                 delete *arestaPtr;
00231                 *arestaPtr = nullptr;
```

```
00232             }
00233             arestaPtr = retornaCelulaMatriz(i, k);
00234             if (*arestaPtr != nullptr) {
00235                 delete *arestaPtr;
00236                 *arestaPtr = nullptr;
00237             }
00238         }
00239 
00240     NodeVertex* newVertices = new NodeVertex[capacidade]();
00241     for (int i = 0, j = 0; i < getOrdem(); i+=1) {
00242         if (i != k) {
00243             newVertices[j].setValue(vertices[i].getValue());
00244             newVertices[j].setGrau(vertices[i].getGrau());
00245             j+=1;
00246         }
00247     }
00248     delete[] vertices;
00249     vertices = newVertices;
00250 
00251     for (int i = k; i < getOrdem()-1; i+=1) {
00252         for (int j = 0; j < getOrdem()-1; j+=1) {
00253             if((*retornaCelulaMatriz(i+1,j+1)) != nullptr && i != j) {
00254                 (*retornaCelulaMatriz(i,j)) = (*retornaCelulaMatriz(i+1,j+1));
00255             } else {
00256                 (*retornaCelulaMatriz(i,j)) = nullptr;
00257             }
00258         }
00259     }
00260 
00261     for (int i = 0; i < getOrdem() - 1; i+=1) {
00262         *retornaCelulaMatriz(i, getOrdem() - 1) = nullptr;
00263         *retornaCelulaMatriz(getOrdem() - 1, i) = nullptr;
00264     }
00265 
00266 
00267 
00268     cout«"Removendo o vértice: "« id«endl;
00269     setOrdem(getOrdem()-1);
00270 
00271 }
00272 
```

6.12 Linked_Vertex.cpp

```
00001 #include "../include/Linked_Vertex.h"
00002 #include "../include/NodeEdge.h"
00003 #include "../include/NodeVertex.h"
00004 #include <iostream>
00005 using namespace std;
00006 
00007 Linked_Vertex::Linked_Vertex() : Linked_list(){};
00008 
00009 Linked_Vertex::~Linked_Vertex()
00010 {
00011     this->limpaNodes();
00012     ultimo = nullptr;
00013 }
00014 
00015 void Linked_Vertex::insereAresta(int origem, int destino, float val)
00016 {
00017     NodeVertex* noOrigem = getNodeById(origem-1);
00018     Linked_list<NodeEdge>* arestas = noOrigem->getArestas();
00019     arestas->insereFinal(destino-1);
00020     arestas->getUltimo()->setPeso(val);
00021 };
00022 
00023 void Linked_Vertex::removeAresta(int i, int j)
00024 {
00025     NodeVertex* no = getNodeById(i-1);
00026     Linked_list<NodeEdge>* arestas = no->getArestas();
00027     NodeEdge* noAresta = (NodeEdge*)arestas->getPrimeiro();
00028     while(noAresta != nullptr && noAresta->getValue() != j-1)
00029     {
00030         noAresta = (NodeEdge*)noAresta->getProx();
00031     }
00032     arestas->removeNode(noAresta);
00033     cout«"Removendo a aresta ("« i«","«j«")"«endl;
00034 
00035 }
00036 
00037 void Linked_Vertex::removeVertice(int id) {
00038     NodeVertex* no = getNodeById(id);
00039     if (no == nullptr){
```

```
00040         return;
00041     }
00042
00043     NodeVertex* current = getPrimeiro();
00044     while (current != nullptr) {
00045         Linked_list<NodeEdge>* arestas = current->getArestas();
00046         NodeEdge* edge = arestas->getPrimeiro();
00047         while (edge != nullptr) {
00048             if (edge->getValue() == id) {
00049                 NodeEdge* auxEdge = edge;
00050                 edge = (NodeEdge*)edge->getProx();
00051                 arestas->removeNode(auxEdge);
00052                 current->setGrau(current->getGrau()-1);
00053             }
00054             else if(edge->getValue() > id)
00055             {
00056                 edge->setValue(edge->getValue()-1);
00057                 edge->setId(edge->getId()-1);
00058                 edge = (NodeEdge*)edge->getProx();
00059             }
00060             else
00061             {
00062                 edge = (NodeEdge*)edge->getProx();
00063             }
00064         }
00065         current = (NodeVertex*)current->getProx();
00066     }
00067
00068     removeNode(no);
00069     cout«"Removendo o vértice: "«id+1«endl;
00070
00071     NodeVertex* p = getPrimeiro();
00072     int newId = 0;
00073     while (p != nullptr) {
00074         p->setId(newId++);
00075         p = (NodeVertex*)p->getProx();
00076     }
00077 }
```

6.13 Node.cpp

```
00001 #include "../include/Node.h"
00002
00003 Node::Node() : id(-1)
00004 {
00005     Node* prox = nullptr;
00006 }
00007
00008 Node::~Node(){}
00009
00010 Node* Node::getProx()
00011 {
00012     return prox;
00013 }
00014
00015 void Node::setProx(Node* prox)
00016 {
00017     this->prox = prox;
00018 }
00019
00020 void Node::setValue(float value)
00021 {
00022     this->value = value;
00023 }
00024
00025 float Node::getValue()
00026 {
00027     return this->value;
00028 }
00029
00030 void Node::setId(int val)
00031 {
00032     this->id = val;
00033 }
00034
00035 int Node::getId()
00036 {
00037     return this->id;
00038 }
```

6.14 NodeEdge.cpp

```
00001 #include "../include/NodeEdge.h"
00002
00003 NodeEdge::NodeEdge() : Node(){
00004     this->peso = 1;
00005 };
00006
00007 NodeEdge::~NodeEdge(){};
00008
00009 float NodeEdge::getPeso()
00010 {
00011     return this->peso;
00012 }
00013
00014 void NodeEdge::setPeso(float val)
00015 {
00016     this->peso = val;
00017 }
```

6.15 NodeVertex.cpp

```
00001 #include "../include/NodeVertex.h"
00002
00003 NodeVertex::NodeVertex() : Node(){
00004     Arestas = new Linked_list<NodeEdge>();
00005     this->grau = 0;
00006 }
00007
00008 NodeVertex::~NodeVertex()
00009 {
00010     delete Arestas;
00011 }
00012
00013 Linked_list<NodeEdge>* NodeVertex::getArestas()
00014 {
00015     return this->Arestas;
00016 }
00017 int NodeVertex::getGrau()
00018 {
00019     return this->grau;
00020 }
00021
00022 void NodeVertex::setGrau(int val)
00023 {
00024     this->grau = val;
00025 }
```

Index