Índice da hierarquia

Hierarquia de classes

Esta lista de heranças está organizada, dentro do possível, por ordem alfabética:

Índice dos componentes

Lista de componentes

Lista de classes, estruturas, uniões e interfaces com uma breve descrição:

Índice dos ficheiros

Lista de ficheiros

Lista de todos os ficheiros com uma breve descrição:

Documentação da classe

Referência à classe Grafo

Classe base para representar um grafo.
`#include <Grafo.h>`

Membros públicos

- **Grafo** (int **numVertices**, bool **direcionado**, bool **ponderadoVertices**, bool **ponderadoArestas**)
  *Construtor da classe **Grafo**.*

- **~Grafo** ()
  *Destrutor da classe **Grafo**.*

- int **n_conexo** ()
  Retorna a quantidade total de componentes conexas do grafo.

- int **get_grau** ()
  Retorna o grau do grafo.

- int **get_ordem** ()
  Retorna a ordem do grafo (numero de vertices do grafo).

- bool **eh_direcionado** ()
  Retorna se o grafo eh direcionado ou nao.

- bool **vertice_ponderado** ()
  Retorna se o grafo possui peso nos vertices ou nao.

- bool **aresta_ponderada** ()
  Retorna se o grafo possui peso nas arestas ou nao.

- bool **eh_completo** ()
  Verifica se o grafo eh completo ou nao.

- void **imprime** ()
  Imprime os atributos do grafo.

- virtual **ListaAdjAresta** * **get_vizinhos** (int id)
- virtual int **get_num_vizinhos** (int id)
- virtual void **dfs** (int v, bool *visitado)
- virtual bool **existe_vertice** (int id)=0
- virtual void **adicionar_vertice** (int id, float peso=0.0)
- virtual void **adicionar_aresta** (int origem, int destino, float peso=1.0)
- virtual void **remover_primeira_aresta** (int id)
- virtual void **remover_vertice** (int id)
- virtual void **remover_aresta** (int origem, int destino)
- virtual int **calcula_menor_dist** (int origem, int destino)=0
- virtual int **calcula_maior_menor_dist** ()=0

Membros públicos estáticos

- static void **carrega_grafo** (**Grafo** *grafo, const string &nomeArquivo)
  Gera um grafo a partir do arquivo grafo.txt.

Atributos Protegidos

- int **numVertices**
  Numero de vertices (ordem) do grafo.

- bool **direcionado**
  Indica se o grafo eh direcionado.

- bool **ponderadoVertices**
  Indica se o grafo possui peso nos vertices.

- bool **ponderadoArestas**
  Indica se o grafo possui peso nas arestas.

Descrição detalhada

Classe base para representar um grafo.

Documentação dos Construtores & Destrutor

Grafo::Grafo (int numVertices, bool direcionado, bool ponderadoVertices, bool ponderadoArestas)

Construtor da classe **Grafo**.

Parâmetros

numVertices	Numero de vertices do grafo.
direcionado	Indica se o grafo eh direcionado.
ponderadoVertices	Indica se o grafo possui peso nos vertices.
ponderadoArestas	Indica se o grafo possui peso nas arestas.

```
18 {
19     this->numVertices = numVertices;
20     this->direcionado = direcionado;
21     this->ponderadoVertices = ponderadoVertices;
22     this->ponderadoArestas = ponderadoArestas;
23 }
```

Grafo::~Grafo ()

Destrutor da classe **Grafo**.

```
29 {
30 }
```

Documentação das funções

virtual void Grafo::adicionar_aresta (int origem, int destino, float peso = 1.0)[inline], [virtual]

Reimplementado em **GrafoLista** (*p. 16*) e **GrafoMatriz** (*p. 25*).

```
98 {};
```

virtual void Grafo::adicionar_vertice (int id, float peso = 0.0)[inline], [virtual]

Reimplementado em **GrafoLista** (*p. 16*) e **GrafoMatriz** (*p. 26*).

```
97 {};
```

bool Grafo::aresta_ponderada ()

Retorna se o grafo possui peso nas arestas ou nao.

Retorna

True se o grafo possui peso nas arestas, false caso contrario.

```
104 {
105     return ponderadoArestas;
106 }
```

virtual int Grafo::calcula_maior_menor_dist () [pure virtual]

Implementado em **GrafoLista** (*p. 17*) e **GrafoMatriz** (*p. 27*).

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

Implementado em **GrafoLista** (*p. 17*) e **GrafoMatriz** (*p. 27*).

void Grafo::carrega_grafo (Grafo * grafo, const string & nomeArquivo) [static]

Gera um grafo a partir do arquivo grafo.txt.

Parâmetros

grafo	Ponteiro para o objeto **Grafo**.
nomeArquivo	Nome do arquivo que contem o grafo.

```
148                                                                 {
149     ifstream arquivo(nomeArquivo);
150     if (!arquivo.is_open()) {
151         cerr << "Erro ao abrir o arquivo: " << nomeArquivo << endl;
152         return;
153     }
154     int numVertices, direcionado, ponderadoVertices, ponderadoArestas;
155     arquivo >> numVertices >> direcionado >> ponderadoVertices >>
ponderadoArestas;
156
157     grafo->direcionado = direcionado;
158     grafo->ponderadoVertices = ponderadoVertices;
159     grafo->ponderadoArestas = ponderadoArestas;
160
161     for(int i = 1; i <= numVertices; i++) {
162         if(ponderadoVertices == 1) {
163             float peso;
164             arquivo >> peso;
165             grafo->adicionar_vertice(i, peso);
166         } else {
167             grafo->adicionar_vertice(i);
168         }
169
170     }
171
172     int origem, destino;
173     while (arquivo >> origem >> destino) {
174         if (ponderadoArestas == 1) {
175             float peso;
176             arquivo >> peso;
177             grafo->adicionar_aresta(origem, destino, peso);
178         } else {
179             grafo->adicionar_aresta(origem, destino);
180         }
181     }
182     arquivo.close();
183 }
```

virtual void Grafo::dfs (int v, bool * visitado) [inline], [virtual]

Reimplementado em **GrafoLista** (*p. 18*) e **GrafoMatriz** (*p. 28*).

```
93 {};
```

bool Grafo::eh_completo ()

Verifica se o grafo eh completo ou nao.

Retorna

True se o grafo eh completo, false caso contrario.

```
113 {
114     for (int i = 1; i <= numVertices; i++) {
115         int numVizinhos = get_num_vizinhos(i);
116         if (numVizinhos != numVertices - 1) {
117             return false;
118         }
119     }
120     return true;
121 }
```

bool Grafo::eh_direcionado ()

Retorna se o grafo eh direcionado ou nao.

Retorna

True se o grafo eh direcionado, false caso contrario.

```
86 {
87     return direcionado;
88 }
```

virtual bool Grafo::existe_vertice (int id) [pure virtual]

Implementado em **GrafoLista** (*p. 19*) e **GrafoMatriz** (*p. 29*).

int Grafo::get_grau ()

Retorna o grau do grafo.

Retorna

Grau do grafo.

```
62 {
63     int grauMax = 0;
64     for (int i = 1 ; i <= numVertices; i++) {
65         if (get_num_vizinhos(i) > grauMax && existe_vertice(i)) {
66             grauMax = get_num_vizinhos(i);
67         }
68     }
69     return grauMax;
70 }
```

virtual int Grafo::get_num_vizinhos (int id) [inline], [virtual]

Reimplementado em **GrafoLista** (*p. 19*) e **GrafoMatriz** (*p. 29*).

```
92 { return 0; };
```

int Grafo::get_ordem ()

Retorna a ordem do grafo (numero de vertices do grafo).

Retorna

Ordem do grafo.

```
77 {
78     return numVertices;
79 }
```

virtual ListaAdjAresta * Grafo::get_vizinhos (int id) [inline], [virtual]

```
91 { return nullptr; };
```

void Grafo::imprime ()

Imprime os atributos do grafo.

```
127 {
128     cout << "Excluindo Noh 1..." << endl;
129     this->remover_vertice(1);
130     cout << "Excluindo primeira aresta do Noh 2..." << endl << endl;
131     this->remover_primeira_aresta(2);
132     cout << "Grau: " << get_grau() << endl;
133     cout << "Ordem: " << numVertices << endl;
134     cout << "Direcionado: " << (eh_direcionado() ? "Sim" : "Nao") << endl;
135     cout << "Componentes conexas: " << n_conexo() << endl;
136     cout << "Vertices ponderados: " << (vertice_ponderado() ? "Sim" : "Nao") <<
endl;
137     cout << "Arestas ponderadas: " << (aresta_ponderada() ? "Sim" : "Nao") << endl;
138     cout << "Completo: " << (eh_completo() ? "Sim" : "Nao") << endl;
139     this->calcula_maior_menor_dist();
140     cout <<
"_____" << endl <<
endl;
141 }
```

int Grafo::n_conexo ()

Retorna a quantidade total de componentes conexas do grafo.

Retorna

Numero de componentes conexas.

```
37 {
38     bool* visitado = new bool[numVertices];
39 
40     for (int i = 1; i <= numVertices; i++) {
41         visitado[i] = false;
42     }
43 
44     int numComponentes = 0;
45 
46     for (int i = 1; i <= numVertices; i++) {
47         if (!visitado[i] && this->existe_vertice(i)) {
48             dfs(i, visitado);
49             numComponentes++;
50         }
51     }
52 
53     delete[] visitado;
54     return numComponentes;
55 }
```

virtual void Grafo::remover_aresta (int origem, int destino) [inline], [virtual]

Reimplementado em **GrafoLista** (*p. 20*) e **GrafoMatriz** (*p. 30*).

```
101 {};
```

virtual void Grafo::remover_primeira_aresta (int id) [inline], [virtual]

Reimplementado em **GrafoLista** (*p. 20*) e **GrafoMatriz** (*p. 31*).

```
99 {};
```

virtual void Grafo::remover_vertice (int id) [inline], [virtual]

Reimplementado em **GrafoLista** (*p. 20*) e **GrafoMatriz** (*p. 31*).

```
100 {};
```

bool Grafo::vertice_ponderado ()

Retorna se o grafo possui peso nos vertices ou nao.

Retorna

True se o grafo possui peso nos vertices, false caso contrario.

```
95 {
96     return ponderadoVertices;
97 }
```

Documentação dos dados membro

bool Grafo::direcionado [protected]

Indica se o grafo eh direcionado.

int Grafo::numVertices [protected]

Numero de vertices (ordem) do grafo.

bool Grafo::ponderadoArestas [protected]

Indica se o grafo possui peso nas arestas.

bool Grafo::ponderadoVertices [protected]

Indica se o grafo possui peso nos vertices.

A documentação para esta classe foi gerada a partir dos seguintes ficheiros:

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

Referência à classe GrafoLista

Classe que representa um grafo utilizando lista de adjacência.

`#include <GrafoLista.h>`

Membros públicos

- **GrafoLista** (int **numVertices**, bool **direcionado**, bool **ponderadoVertices**, bool **ponderadoArestas**)
  *Construtor da classe **GrafoLista**.*

- **~GrafoLista** ()
  *Destrutor da classe **GrafoLista**.*

- int **get_num_vizinhos** (int id) override
  Retorna o numero de vizinhos de um vertice do grafo.

- void **dfs** (int id, bool *visitado) override
  Realiza a busca em profundidade no grafo.

- bool **existe_vertice** (int id) override
  Verifica se um vertice existe no grafo.

- void **adicionar_vertice** (int id, float peso=0.0) override
  Adiciona um vertice no grafo.

- void **adicionar_aresta** (int origem, int destino, float peso=1.0) override
  Adiciona uma aresta no grafo.

- void **remover_vertice** (int id) override
  Remove um vertice do grafo.

- void **remover_aresta** (int origem, int destino) override
  Remove uma aresta do grafo.

- void **remover_primeira_aresta** (int id) override
  Remove a primeira aresta de um vertice.

- int **calcula_menor_dist** (int origem, int destino)
  Calcula a menor distancia entre dois vertices.

- int **calcula_maior_menor_dist** ()
  Calcula a maior entre as menores distancias de um grafo.

- void **imprimeGrafoLista** ()
  Imprime as informacoes do grafo.

- void **imprimeListaAdj** ()
  Imprime a lista de adjacencia.

Membros públicos herdados de Grafo

- **Grafo** (int **numVertices**, bool **direcionado**, bool **ponderadoVertices**, bool **ponderadoArestas**)
  *Construtor da classe **Grafo**.*

- **~Grafo** ()
  *Destrutor da classe **Grafo**.*

- int **n_conexo** ()
  Retorna a quantidade total de componentes conexas do grafo.

- int **get_grau** ()
  Retorna o grau do grafo.

- int **get_ordem** ()
  Retorna a ordem do grafo (numero de vertices do grafo).

- bool **eh_direcionado** ()
  Retorna se o grafo eh direcionado ou nao.

- bool **vertice_ponderado** ()
  Retorna se o grafo possui peso nos vertices ou nao.

- bool **aresta_ponderada** ()
  Retorna se o grafo possui peso nas arestas ou nao.

- bool **eh_completo** ()
  Verifica se o grafo eh completo ou nao.

- void **imprime** ()

Imprime os atributos do grafo.

- virtual **ListaAdjAresta * get_vizinhos** (int id)

Atributos Protegidos

- ListaAdjVertice * **listaAdjVertices**
  Lista encadeada de vertices.

Atributos Protegidos herdados de Grafo

- int **numVertices**
  Numero de vertices (ordem) do grafo.

- bool **direcionado**
  Indica se o grafo eh direcionado.

- bool **ponderadoVertices**
  Indica se o grafo possui peso nos vertices.

- bool **ponderadoArestas**
  Indica se o grafo possui peso nas arestas.

Outros membros herdados

Membros públicos estáticos herdados de Grafo

- static void **carrega_grafo** (**Grafo** *grafo, const string &nomeArquivo)
  Gera um grafo a partir do arquivo grafo.txt.

Descrição detalhada

Classe que representa um grafo utilizando lista de adjacência.

Documentação dos Construtores & Destrutor

GrafoLista::GrafoLista (int numVertices, bool direcionado, bool ponderadoVertices, bool ponderadoArestas)

Construtor da classe **GrafoLista**.

Parâmetros

numVertices	Numero de vertices do grafo.
direcionado	Indica se o grafo eh direcionado.
ponderadoVertices	Indica se o grafo possui peso nos vertices.
ponderadoArestas	Indica se o grafo possui peso nas arestas.

```
15
: Grafo(numVertices, direcionado, ponderadoVertices, ponderadoArestas)
16 {
17     // Inicializa a lista de adjacencia
18     this->listaAdjVertices = new ListaAdjVertice();
19 }
```

GrafoLista::~GrafoLista ()

Destrutor da classe **GrafoLista**.

```
25 {
26     // Libera a memoria alocada para os vertices
27     delete this->listaAdjVertices;
28 }
```

Documentação das funções

void GrafoLista::adicionar_aresta (int origem, int destino, float peso = 1.0)[override], [virtual]

Adiciona uma aresta no grafo.

Adiciona uma aresta ao grafo.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.
peso	Peso da aresta (opcional).

Reimplementado de **Grafo** (*p. 7*).

```
62                                                                          {
63     // Verifica se o vertice de origem existe
64     if(!existe_vertice(origem)){
65         cout << "Erro: Vertice " << origem << " nao existe!" << endl;
66         return;
67     }
68     // Verifica se o vertice de destino existe
69     if(!existe_vertice(destino)){
70         cout << "Erro: Vertice " << destino << " nao existe!" << endl;
71         return;
72     }
73     // Verifica se a aresta eh um self-loop
74     if(origem == destino){
75         cout << "Erro: Nao eh possivel adicionar aresta. Origem e destino iguais!"
<< endl;
76         return;
77     }
78
79     // Adiciona a aresta
80     this->listaAdjVertices->adicionar_aresta(origem, destino, peso);
81     if(!direcionado){
82         this->listaAdjVertices->adicionar_aresta(destino, origem, peso);
83     }
84 }
```

void GrafoLista::adicionar_vertice (int id, float peso = 0.0)[override], [virtual]

Adiciona um vertice no grafo.

Adiciona um vertice ao grafo.

Parâmetros

id	Identificador do vertice.
peso	Peso do vertice (opcional).

Reimplementado de **Grafo** (*p. 7*).

```
44                                                      {
45      // Verifica se o vertice ja existe
46      if(existe_vertice(id)){
47          cout << "Erro: Vertice " << id << " ja existe!" << endl;
48          return;
49      }
50
51      // Adiciona o vertice
52      this->listaAdjVertices->adicionar_vertice(id, peso);
53      this->numVertices++;
54  }
```

int GrafoLista::calcula_maior_menor_dist () `[virtual]`

Calcula a maior entre as menores distancias de um grafo.

Retorna

Maior entre as menores distancias.

Implementa **Grafo** (*p. 7*).

```
190                                             {
191      int maiorMenorDist = 0;
192      int verticeOrigem = -1, verticeDestino = -1;
193
194      for (int i = 1; i <= numVertices; i++) {
195          for (int j = 1; j <= numVertices; j++) {
196              if (i != j) {
197                  int menorDist = calcula_menor_dist(i, j);
198                  if (menorDist != -1 && menorDist > maiorMenorDist) {
199                      maiorMenorDist = menorDist;
200                      verticeOrigem = i;
201                      verticeDestino = j;
202                  }
203              }
204          }
205      }
206
207      if (verticeOrigem != -1 && verticeDestino != -1) {
208          cout << "Maior menor distancia: (" << verticeOrigem << "-" << verticeDestino
<< ") = " << maiorMenorDist << endl;
209      } else {
210          cout << "Nao ha caminhos validos no grafo." << endl;
211      }
212
213      return maiorMenorDist;
214  }
```

int GrafoLista::calcula_menor_dist (int origem, int destino) `[virtual]`

Calcula a menor distancia entre dois vertices.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.

Retorna

Menor distancia entre os vertices.

Implementa **Grafo** (*p. 7*).

```
147                                                                 {
148     const int INF = 1000000;
149     int dist[numVertices + 1];
150     bool visitado[numVertices + 1];
151 
152     for (int i = 1; i <= numVertices; i++) {
153         dist[i] = INF;
154         visitado[i] = false;
155     }
156 
157     dist[origem] = 0;
158 
159     for (int i = 1; i <= numVertices; i++) {
160         int u = -1;
161         for (int j = 1; j <= numVertices; j++) {
162             if (!visitado[j] && (u == -1 || dist[j] < dist[u])) {
163                 u = j;
164             }
165         }
166 
167         if (dist[u] == INF) {
168             break;
169         }
170 
171         visitado[u] = true;
172         NoAresta* atual =
listaAdjVertices->getVertice(u)->getArestas()->getCabeca();
173         while (atual != nullptr) {
174             int v = atual->getDestino();
175             float peso = atual->getPeso();
176             if (dist[u] + peso < dist[v]) {
177                 dist[v] = dist[u] + peso;
178             }
179             atual = atual->getProximo();
180         }
181     }
182 
183     return (dist[destino] == INF) ? -1 : dist[destino];
184 }
```

void GrafoLista::dfs (int id, bool * visitado) `[override]`, `[virtual]`

Realiza a busca em profundidade no grafo.

Parâmetros

id	Identificador do vertice inicial.
visitado	Vetor de vertices visitados.

Reimplementado de **Grafo** (*p. 8*).

```
230                                             {
231     if(listaAdjVertices->getVertice(id) == nullptr){
232         return;
233     }
234     visitado[id] = true;
235     NoAresta* atual =
this->listaAdjVertices->getVertice(id)->getArestas()->getCabeca();
236     while(atual != nullptr){
237         if(!visitado[atual->getDestino()]){
238             dfs(atual->getDestino(), visitado);
239         }
240         atual = atual->getProximo();
241     }
242 }
```

bool GrafoLista::existe_vertice (int id) [override], [virtual]

Verifica se um vertice existe no grafo.

Parâmetros

id	Identificador do vertice.

Retorna

True se o vertice existe, false caso contrario.

Implementa **Grafo** (*p. 9*).

```
35                                          {
36     return this->listaAdjVertices->getVertice(id) != nullptr;
37 }
```

int GrafoLista::get_num_vizinhos (int id) [override], [virtual]

Retorna o numero de vizinhos de um vertice do grafo.

Retorna a quantidade de vizinhos de um vertice.

Parâmetros

id	Identificador do vertice.

Retorna

Numero de vizinhos do vertice.

Reimplementado de **Grafo** (*p. 9*).

```
221                                           {
222     return this->listaAdjVertices->getVertice(id)->getNumVizinhos();
223 }
```

void GrafoLista::imprimeGrafoLista ()

Imprime as informacoes do grafo.

Imprime os atributos do grafo.

```
255                                     {
256     cout <<
"_____________________________________________________________________" << endl;
257     cout << endl << "--- Grafo Lista ---" << endl;
258     cout <<
"_____________________________________________________________________" << endl <<
endl;
259     // Imprime as informacoes do grafo
260     imprime();
261 }
```

void GrafoLista::imprimeListaAdj ()

Imprime a lista de adjacencia.

```
247                                   {
248     // Imprime a lista de adjacencia
249     listaAdjVertices->imprimir();
250 }
```

void GrafoLista::remover_aresta (int origem, int destino) [override], [virtual]

Remove uma aresta do grafo.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.

Reimplementado de **Grafo** (*p. 10*).

```
91                                                      {
92      // Verifica se o vertice de origem existe
93      if(listaAdjVertices->getVertice(origem) == nullptr){
94          cout << "Erro: Vertice " << origem << " nao existe!" << endl;
95          return;
96      }
97      // Verifica se o vertice de destino existe
98      if(listaAdjVertices->getVertice(destino) == nullptr){
99          cout << "Erro: Vertice " << destino << " nao existe!" << endl;
100          return;
101      }
102
103      // Remove a aresta
104      this->listaAdjVertices->remover_aresta(origem, destino);
105      if(!direcionado){
106          this->listaAdjVertices->remover_aresta(destino, origem);
107      }
108 }
```

void GrafoLista::remover_primeira_aresta (int id) [override], [virtual]

Remove a primeira aresta de um vertice.

Parâmetros

id	Identificador do vertice.

Reimplementado de **Grafo** (*p. 11*).

```
114                                                 {
115      // Verifica se o vertice existe
116      if(listaAdjVertices->getVertice(id) == nullptr){
117          cout << "Erro: Vertice " << id << " nao existe!" << endl;
118          return;
119      }
120
121      // Remove a primeira aresta
122      this->listaAdjVertices->remover_primeira_aresta(id);
123 }
```

void GrafoLista::remover_vertice (int id) [override], [virtual]

Remove um vertice do grafo.

Parâmetros

id	Identificador do vertice.

Reimplementado de **Grafo** (*p. 11*).

```
129                                         {
130      // Verifica se o vertice existe
131      if(!existe_vertice(id)){
```

```
132         cout << "Erro: Vertice " << id << " nao existe!" << endl;
133         return;
134     }
135 
136     // Remove o vertice
137     this->listaAdjVertices->remover_vertice(id);
138     this->numVertices--;
139 }
```

Documentação dos dados membro

ListaAdjVertice* GrafoLista::listaAdjVertices `[protected]`

Lista encadeada de vertices.

A documentação para esta classe foi gerada a partir dos seguintes ficheiros:

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

Referência à classe GrafoMatriz

Classe que representa um grafo utilizando matriz de adjacência.

`#include <GrafoMatriz.h>`

Membros públicos

- **GrafoMatriz** (int **numVertices**, bool **direcionado**, bool **ponderadoVertices**, bool **ponderadoArestas**)
  *Construtor da classe **GrafoMatriz**.*

- **~GrafoMatriz** ()
  *Destrutor da classe **GrafoMatriz**.*

- int **grauVertice** (int vertice)
  Retorna o grau de um vertice do grafo.

- int **get_num_vizinhos** (int id) override
  Retorna o numero de vizinhos de um vertice do grafo.

- void **dfs** (int id, bool *visitado) override
  Realiza a busca em profundidade no grafo.

- bool **existe_vertice** (int id) override
  Verifica se um vertice existe no grafo.

- void **adicionar_vertice** (int id, float peso=0.0) override
  Adiciona um vertice no grafo.

- void **adicionar_aresta** (int origem, int destino, float peso=1.0) override
  Adiciona uma aresta no grafo.

- void **remover_vertice** (int id) override
  Remove um vertice do grafo.

- void **remover_aresta** (int origem, int destino) override
  Remove uma aresta do grafo.

- void **remover_primeira_aresta** (int id) override
  Remove a primeira aresta de um vertice.

- int **calcula_menor_dist** (int origem, int destino) override
  Calcula a menor distancia entre dois vertices.

- int **calcula_maior_menor_dist** () override
  Calcula a maior entre as menores distancias de um grafo.

- void **imprimirMatrizAdj** ()
  Imprime a matriz de adjacencia.

- void **imprimeGrafoMatriz** ()
  Imprime os atributos do grafo.

Membros públicos herdados de Grafo

- **Grafo** (int **numVertices**, bool **direcionado**, bool **ponderadoVertices**, bool **ponderadoArestas**)
  *Construtor da classe **Grafo**.*

- **~Grafo** ()
  *Destrutor da classe **Grafo**.*

- int **n_conexo** ()
  Retorna a quantidade total de componentes conexas do grafo.

- int **get_grau** ()
  Retorna o grau do grafo.

- int **get_ordem** ()
  Retorna a ordem do grafo (numero de vertices do grafo).

- bool **eh_direcionado** ()
  Retorna se o grafo eh direcionado ou nao.

- bool **vertice_ponderado** ()
  Retorna se o grafo possui peso nos vertices ou nao.

- bool **aresta_ponderada** ()
  Retorna se o grafo possui peso nas arestas ou nao.

- bool **eh_completo** ()

Verifica se o grafo eh completo ou nao.

- void **imprime** ()
  Imprime os atributos do grafo.

- virtual **ListaAdjAresta** * **get_vizinhos** (int id)

Atributos Protegidos

- int ** **matrizAdj**
  Matriz de adjacencia.

- int **tamanhoMatriz**
  Numero MXM vertices.

Atributos Protegidos herdados de Grafo

- int **numVertices**
  Numero de vertices (ordem) do grafo.

- bool **direcionado**
  Indica se o grafo eh direcionado.

- bool **ponderadoVertices**
  Indica se o grafo possui peso nos vertices.

- bool **ponderadoArestas**
  Indica se o grafo possui peso nas arestas.

Outros membros herdados

Membros públicos estáticos herdados de Grafo

- static void **carrega_grafo** (**Grafo** *grafo, const string &nomeArquivo)
  Gera um grafo a partir do arquivo grafo.txt.

Descrição detalhada

Classe que representa um grafo utilizando matriz de adjacência.

Documentação dos Construtores & Destrutor

GrafoMatriz::GrafoMatriz (int numVertices, bool direcionado, bool ponderadoVertices, bool ponderadoArestas)

Construtor da classe **GrafoMatriz**.

Parâmetros

numVertices	Numero de vertices do grafo.
direcionado	Indica se o grafo eh direcionado.
ponderadoVertices	Indica se o grafo possui peso nos vertices.
ponderadoArestas	Indica se o grafo possui peso nas arestas.

```
17
: Grafo(numVertices, direcionado, ponderadoVertices, ponderadoArestas)
18 {
19     tamanhoMatriz = 10;
20
21     if (numVertices <= 0)
22     {
23         numVertices = 1;
24     }
25
26     while (tamanhoMatriz < numVertices)
27     {
28         tamanhoMatriz *= 2;
29     }
30
31     matrizAdj = new int *[tamanhoMatriz];
32     for (int i = 0; i < tamanhoMatriz; i++)
33     {
34         matrizAdj[i] = new int[tamanhoMatriz];
35         for (int j = 0; j < tamanhoMatriz; j++)
36         {
37             matrizAdj[i][j] = 0; // Inicializa com 0 (sem aresta)
38         }
39     }
40 }
```

GrafoMatriz::~GrafoMatriz ()

Destrutor da classe **GrafoMatriz**.

```
46 {
47     if (matrizAdj != nullptr)
48     { // Verifica se ha memoria alocada
49         for (int i = 0; i < numVertices; i++)
50         {
51             delete[] matrizAdj[i];
52         }
53         delete[] matrizAdj;
54         matrizAdj = nullptr; // Evita ponteiro danificado
55     }
56 }
```

Documentação das funções

void GrafoMatriz::adicionar_aresta (int origem, int destino, float peso = 1.0)[override], [virtual]

Adiciona uma aresta no grafo.

Adiciona uma aresta ao grafo.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.
peso	Peso da aresta (opcional).

Reimplementado de **Grafo** (*p. 7*).

```
201 {
202     if (origem < 0 || origem > numVertices || destino < 0 || destino > numVertices)
203     {
204         cout << "Erro: indices de origem ou destino invalidos.\n";
205         return;
206     }
207 
208     if (ponderadoArestas)
209     {
210         matrizAdj[origem][destino] = peso;
211         if (!direcionado)
212         {
213             matrizAdj[destino][origem] = peso;
214         }
215     }
216     else
217     {
218         matrizAdj[origem][destino] = 1;
219         if (!direcionado)
220         {
221             matrizAdj[destino][origem] = 1;
222         }
223     }
224 }
```

void GrafoMatriz::adicionar_vertice (int id, float peso = 0.0)[override], [virtual]

Adiciona um vertice no grafo.

Adiciona um vertice ao grafo.

Parâmetros

id	Identificador do vertice.
peso	Peso do vertice (opcional).

Reimplementado de **Grafo** (*p. 7*).

```
64 {
65     int novoNumVertices = numVertices + 1;
66     numVertices = novoNumVertices;
67 
68     if (novoNumVertices >= tamanhoMatriz)
69     {
70         cout << "Aumentando matriz de adjacencia...\n";
71         int antigoTamanhoMatriz = tamanhoMatriz;
72         tamanhoMatriz = max(tamanhoMatriz * 2, novoNumVertices); // Expansao mais
segura
73         int **novaMatriz = new int *[tamanhoMatriz];
74 
75         for (int i = 0; i < tamanhoMatriz; i++)
76         {
77             novaMatriz[i] = new int[tamanhoMatriz](); // Inicializa com 0
78         }
79 
80         // Copia os valores da matriz antiga para a nova matriz
81         for (int i = 0; i < antigoTamanhoMatriz; i++)
82         {
83             for (int j = 0; j < antigoTamanhoMatriz; j++)
84             {
85                 novaMatriz[i][j] = matrizAdj[i][j];
86             }
87         }
88 
89         // Libera a matriz antiga corretamente
90         for (int i = 0; i < antigoTamanhoMatriz; i++)
91         {
92             delete[] matrizAdj[i];
93         }
94         delete[] matrizAdj;
95 
96         matrizAdj = novaMatriz;
97     }
98 }
```

int GrafoMatriz::calcula_maior_menor_dist () [override], [virtual]

Calcula a maior entre as menores distancias de um grafo.

Retorna

Maior entre as menores distancias.

Implementa **Grafo** (*p. 7*).

```
320                                             {
321     int maiorMenorDist = 0;
322     int verticeOrigem = -1;
323     int verticeDestino = -1;
324 
325     for (int i = 1; i <= numVertices; i++) {
326         for (int j = 1; j <= numVertices; j++) {
327             if (i != j) {
328                 int menorDist = calcula_menor_dist(i, j);
329                 if (menorDist != 1000000 && menorDist > maiorMenorDist) {
330                     maiorMenorDist = menorDist;
331                     verticeOrigem = i;
332                     verticeDestino = j;
333                 }
334             }
335         }
336     }
337 
338     if (verticeOrigem != -1 && verticeDestino != -1) {
339         cout << "Maior menor distancia: (" << verticeOrigem << "-" << verticeDestino
<< ") = " << maiorMenorDist << endl;
340     } else {
341         cout << "Nao ha caminhos validos no grafo." << endl;
342     }
343 
344     return maiorMenorDist;
345 }
```

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

Calcula a menor distancia entre dois vertices.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.

Retorna

Menor distancia entre os vertices.

Implementa **Grafo** (*p. 7*).

```
233 {
234     const int INF = 1000000; // Valor grande para representar infinito
235     int *dist = new int[numVertices + 1];
236     int *prev = new int[numVertices + 1];
237     bool *visitado = new bool[numVertices + 1];
238 
239     for (int i = 1; i <= numVertices; i++)
240     {
241         dist[i] = INF;
242         prev[i] = -1;
243         visitado[i] = false;
244     }
245 
246     dist[origem] = 0;
247 
248     // Loop principal do algoritmo de Dijkstra
```

```
249     for (int i = 1; i <= numVertices; i++)
250     {
251         int u = -1;
252 
253         // Encontra o vertice nao visitado com a menor distancia
254         for (int j = 1; j <= numVertices; j++)
255         {
256             if (!visitado[j] && (u == -1 || dist[j] < dist[u]))
257             {
258                 u = j;
259             }
260         }
261 
262         // Se a menor distancia e infinita, todos os vertices restantes sao
inacessiveis
263         if (dist[u] == INF)
264         {
265             break;
266         }
267 
268         visitado[u] = true;
269 
270         // Atualiza as distancias dos vizinhos do vertice atual
271         for (int v = 1; v <= numVertices; v++)
272         {
273             if (matrizAdj[u][v] != 0 && dist[u] + matrizAdj[u][v] < dist[v])
274             {
275                 dist[v] = dist[u] + matrizAdj[u][v];
276                 prev[v] = u;
277             }
278         }
279     }
280 
281     int menorDist = dist[destino];
282     // Verifica se ha um caminho ate o destino
283     if (menorDist == INF)
284     {
285         // Nao ha caminho entre origem e destino
286     }
287     else
288     {
289         // Caminho encontrado
290         int* caminho = new int[numVertices];
291         int count = 0;
292         for (int at = destino; at != -1; at = prev[at])
293         {
294             caminho[count++] = at;
295         }
296 
297         // Imprime o caminho na ordem correta
298         for (int i = count - 1; i >= 0; i--) {
299             // std::cout << caminho[i] + 1;
300             if (i > 0) {
301                 // std::cout << " -> ";
302             }
303         }
304         // std::cout << std::endl;
305 
306         delete[] caminho;
307     }
308 
309     delete[] dist;
310     delete[] prev;
311     delete[] visitado;
312 
313     return menorDist;
314 }
```

void GrafoMatriz::dfs (int id, bool * visitado) `[override]`, `[virtual]`

Realiza a busca em profundidade no grafo.

Parâmetros

id	Identificador do vertice inicial.
visitado	Vetor de vertices visitados.

Reimplementado de **Grafo** (*p. 8*).

```
376 {
377     if (id < 0 || id > numVertices || visitado[id])
378     {
379         return; // Verifica se o vertice eh valido ou ja foi visitado
380     }
381 
382     visitado[id] = true; // Marca o vertice como visitado
383 
384     for (int i = 0; i <= numVertices; i++)
385     {
386         if (matrizAdj[id][i] != 0 && !visitado[i])
387         { // Se houver uma aresta e o vertice nao foi visitado
388             dfs(i, visitado);
389         }
390     }
391 }
```

bool GrafoMatriz::existe_vertice (int id) `[override]`, `[virtual]`

Verifica se um vertice existe no grafo.

Parâmetros

id	Identificador do vertice.

Retorna

True se o vertice existe, false caso contrario.

Implementa **Grafo** (*p. 9*).

```
399 {
400     return (id >= 0 && id <= numVertices);
401 }
```

int GrafoMatriz::get_num_vizinhos (int id) `[override]`, `[virtual]`

Retorna o numero de vizinhos de um vertice do grafo.

Retorna a quantidade de vizinhos de um vertice.

Parâmetros

id	Identificador do vertice.

Retorna

Numero de vizinhos do vertice.

Reimplementado de **Grafo** (*p. 9*).

```
353 {
354     if (id < 0 || id > numVertices)
355     {
356         return 0; // Retorna 0 se o ID do vertice for invalido
357     }
358 
359     int contador = 0;
360     for (int i = 0; i <= numVertices; i++)
361     {
362         if (matrizAdj[id][i] != 0)
363         { // Se houver uma aresta entre os vertices
364             contador++;
365         }
```

```
366     }
367     return contador;
368 }
```

int GrafoMatriz::grauVertice (int vertice)

Retorna o grau de um vertice do grafo.

Parâmetros

vertice	Identificador do vertice.

Retorna

Grau do vertice.

void GrafoMatriz::imprimeGrafoMatriz ()

Imprime os atributos do grafo.

```
437 {
438     cout <<
"_____________________________________________________________________" << endl;
439     cout << endl
440          << "--- Grafo Matriz ---" << endl;
441     cout <<
"_____________________________________________________________________" << endl
442          << endl;
443     imprime();
444 }
```

void GrafoMatriz::imprimirMatrizAdj ()

Imprime a matriz de adjacencia.

```
407 {
408     cout <<
"_____________________________________________________________________" << endl;
409     cout << endl << "--- Matriz de Adjacencia ---" << endl << endl;
410
411     // Imprime o cabecalho dos indices dos vertices
412     cout << "  V";
413     for (int i = 0; i < tamanhoMatriz; i++)
414     {
415         cout << std::setw(3) << i << " ";
416     }
417     cout << endl;
418
419     for (int i = 0; i < tamanhoMatriz; i++)
420     {
421         // Imprime o indice do vertice na lateral esquerda
422         cout << std::setw(3) << i;
423
424         for (int j = 0; j < tamanhoMatriz; j++)
425         {
426             std::cout << std::setw(3) << matrizAdj[i][j] << " ";
427         }
428         std::cout << std::endl;
429     }
430     cout <<
"_____________________________________________________________________" << endl;
431 }
```

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

Remove uma aresta do grafo.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.

Reimplementado de **Grafo** (*p. 10*).

```
154 {
155     if (origem < 0 || origem >= numVertices || destino < 0 || destino >= numVertices)
156     {
157         cout << "Erro: indices de origem ou destino invalidos.\n";
158         return;
159     }
160 
161     matrizAdj[origem][destino] = 0;
162     if (!direcionado)
163     {
164         matrizAdj[destino][origem] = 0;
165     }
166 }
```

void GrafoMatriz::remover_primeira_aresta (int id) `[override]`, `[virtual]`

Remove a primeira aresta de um vertice.

Parâmetros

id	Identificador do vertice.

Reimplementado de **Grafo** (*p. 11*).

```
173 {
174     if (id < 0 || id > numVertices)
175     {
176         cout << "Erro: indice de vertice invalido." << endl;
177         return;
178     }
179 
180     for (int i = 0; i <= numVertices; i++)
181     {
182         if (matrizAdj[id][i] != 0)
183         {
184             matrizAdj[id][i] = 0;
185             if (!direcionado)
186             {
187                 matrizAdj[i][id] = 0;
188             }
189             return;
190         }
191     }
192 }
```

void GrafoMatriz::remover_vertice (int id) `[override]`, `[virtual]`

Remove um vertice do grafo.

Parâmetros

id	Identificador do vertice.

Reimplementado de **Grafo** (*p. 11*).

```
105 {
106     if (id < 0 || id >= numVertices)
```

```
107     {
108         cout << "Erro: indice de vertice invalido.\n";
109         return;
110     }
111 
112     int novoNumVertices = numVertices - 1;
113     int **novaMatriz = new int *[tamanhoMatriz];
114 
115     for (int i = 0; i < tamanhoMatriz; i++)
116     {
117         novaMatriz[i] = new int[tamanhoMatriz]();
118     }
119 
120     // Copia os valores da matriz antiga para a nova, excluindo o vertice removido
121     for (int i = 0, ni = 0; i < tamanhoMatriz; i++)
122     {
123         if (i == id)
124             continue;
125 
126         for (int j = 0, nj = 0; j < tamanhoMatriz; j++)
127         {
128             if (j == id)
129                 continue;
130 
131             novaMatriz[ni][nj] = matrizAdj[i][j];
132             nj++;
133         }
134         ni++;
135     }
136 
137     // Libera a matriz antiga
138     for (int i = 0; i < numVertices; i++)
139     {
140         delete[] matrizAdj[i];
141     }
142     delete[] matrizAdj;
143 
144     matrizAdj = novaMatriz;
145     numVertices = novoNumVertices;
146 }
```

Documentação dos dados membro

int** GrafoMatriz::matrizAdj [protected]

Matriz de adjacencia.

int GrafoMatriz::tamanhoMatriz [protected]

Numero MXM vertices.

A documentação para esta classe foi gerada a partir dos seguintes ficheiros:

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

Referência à classe ListaAdjAresta

Classe que representa uma lista de adjacência de arestas.
`#include <ListaAdjAresta.h>`

Membros públicos

- **ListaAdjAresta** ()
  *Construtor da classe **ListaAdjAresta**.*

- **~ListaAdjAresta** ()
  *Destrutor da classe **ListaAdjAresta**.*

- **NoAresta * getCabeca** ()
  Retorna a cabeca da lista de adjacencia de arestas.

- int **getNumVerticesVizinhos** ()
  Retorna a quantidade de vertices vizinhos.

- void **adicionar_aresta** (int origem, int destino, float peso)
  Adiciona uma aresta à lista.

- void **remover_aresta** (int origem, int destino)
  Remove uma aresta da lista.

- void **remover_primeira_aresta** ()
  Remove a primeira aresta da lista.

- **ListaAdjAresta** ()
  *Construtor da classe **ListaAdjAresta**.*

- **~ListaAdjAresta** ()
  *Destrutor da classe **ListaAdjAresta**.*

- **NoAresta * getCabeca** ()
  Retorna a cabeca da lista de adjacencia de arestas.

- int **getNumVerticesVizinhos** ()
  Retorna a quantidade de vertices vizinhos.

- void **adicionar_aresta** (int origem, int destino, float peso)
  Adiciona uma aresta à lista.

- void **remover_aresta** (int origem, int destino)
  Remove uma aresta da lista.

- void **remover_primeira_aresta** ()
  Remove a primeira aresta da lista.

Descrição detalhada

Classe que representa uma lista de adjacência de arestas.

Documentação dos Construtores & Destrutor

ListaAdjAresta::ListaAdjAresta ()

Construtor da classe **ListaAdjAresta**.

```
12                                  {
13     this->cabeca = nullptr;
14 }
```

ListaAdjAresta::~ListaAdjAresta ()

Destrutor da classe **ListaAdjAresta**.

```
19                                   {
20     // Desaloca a memoria para cada no da lista de arestas
21     NoAresta* atual = this->cabeca;
22     while (atual != nullptr) {
23         NoAresta* next = atual->getProximo();
24         delete atual;
25         atual = next;
26     }
27 }
```

ListaAdjAresta::ListaAdjAresta ()

Construtor da classe **ListaAdjAresta**.

ListaAdjAresta::~ListaAdjAresta ()

Destrutor da classe **ListaAdjAresta**.

Documentação das funções

void ListaAdjAresta::adicionar_aresta (int origem, int destino, float peso)

Adiciona uma aresta à lista.

Adiciona uma aresta a lista.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.
peso	Peso da aresta.

```
46                                                                          {
47     // Verifica se a aresta ja existe
```

```
48      NoAresta* atual = this->cabeca;
49      while (atual != nullptr) {
50          if (atual->getOrigem() == origem && atual->getDestino() == destino) {
51              return;
52          }
53          atual = atual->getProximo();
54      }
55
56      // Adiciona uma nova aresta
57      NoAresta* novaAresta = new NoAresta(origem, destino, peso);
58      novaAresta->setProximo(this->cabeca);
59      this->cabeca = novaAresta;
60  }
```

void ListaAdjAresta::adicionar_aresta (int origem, int destino, float peso)

Adiciona uma aresta à lista.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.
peso	Peso da aresta.

NoAresta * ListaAdjAresta::getCabeca ()

Retorna a cabeca da lista de adjacencia de arestas.

Retorna o primeiro no da lista (primeira aresta do vertice).

Retorna

Ponteiro para o primeiro no da lista.

```
36                                      {
37      return this->cabeca;
38  }
```

NoAresta * ListaAdjAresta::getCabeca ()

Retorna a cabeca da lista de adjacencia de arestas.

Retorna

Ponteiro para o primeiro no da lista.

int ListaAdjAresta::getNumVerticesVizinhos ()

Retorna a quantidade de vertices vizinhos.

Retorna o tamanho da lista / numero de vertices vizinhos / grau do vertice.

Retorna

Numero de vertices vizinhos.

```
110                                             {
111     int tamanho = 0;
112     NoAresta* atual = this->cabeca;
113     while (atual != nullptr) {
114         tamanho++;
115         atual = atual->getProximo();
116     }
117     return tamanho;
```

```
118 }
```

int ListaAdjAresta::getNumVerticesVizinhos ()

Retorna a quantidade de vertices vizinhos.

Retorna

Numero de vertices vizinhos.

void ListaAdjAresta::remover_aresta (int origem, int destino)

Remove uma aresta da lista.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.

```
67                                                          {
68     NoAresta* atual = this->cabeca;
69     NoAresta* anterior = nullptr;
70     while (atual != nullptr) {
71         if (atual->getOrigem() == origem && atual->getDestino() == destino) {
72             if (anterior == nullptr) {
73                 this->cabeca = atual->getProximo();
74             } else {
75                 anterior->setProximo(atual->getProximo());
76             }
77             delete atual;
78             return;
79         }
80         anterior = atual;
81         atual = atual->getProximo();
82     }
83 }
```

void ListaAdjAresta::remover_aresta (int origem, int destino)

Remove uma aresta da lista.

Parâmetros

origem	Identificador do vertice de origem.
destino	Identificador do vertice de destino.

void ListaAdjAresta::remover_primeira_aresta ()

Remove a primeira aresta da lista.

```
88                                                  {
89     if (this->cabeca == nullptr) {
90         cout << "Vertice nao possui arestas" << endl;
91         return;
92     }
93
94     NoAresta* atual = this->cabeca;
95     NoAresta* menor = this->cabeca;
96
97     while (atual != nullptr) {
98         if (atual->getDestino() < menor->getDestino()) {
```

```
99              menor = atual;
100         }
101         atual = atual->getProximo();
102     }
103     remover_aresta(menor->getOrigem(), menor->getDestino());
104 }
```

void ListaAdjAresta::remover_primeira_aresta ()

Remove a primeira aresta da lista.

A documentação para esta classe foi gerada a partir dos seguintes ficheiros:

- TrabalhoGrafos/include/**ListaAdjAresta.h**
- TrabalhoGrafos/include/**ListaAdjVertice.h**
- TrabalhoGrafos/src/**ListaAdjAresta.cpp**

Referência à classe NoAresta

Classe que representa um nó de aresta em uma lista de adjacência.
`#include <NoAresta.h>`

Membros públicos

- **NoAresta** (int idVerticeOrigem, int idVerticeDestino, float peso)
  *Construtor da classe **NoAresta**.*

- **~NoAresta** ()
  *Destrutor da classe **NoAresta**.*

- int **getOrigem** ()
  Retorna o vertice de origem.

- int **getDestino** ()
  Retorna o vertice de destino.

- float **getPeso** ()
  Retorna o peso da aresta.

- **NoAresta * getProximo** ()
  Retorna o proximo no aresta.

- void **setProximo** (**NoAresta** *proximo)
  Define o proximo no aresta.

- void **setVerticeOrigem** (int novoId)
  Define o identificador do vertice de origem.

- void **setVerticeDestino** (int novoId)
  Define o identificador do vertice de destino.

Descrição detalhada

Classe que representa um nó de aresta em uma lista de adjacência.

Documentação dos Construtores & Destrutor

NoAresta::NoAresta (int idVerticeOrigem, int idVerticeDestino, float peso)

Construtor da classe **NoAresta**.

Parâmetros

idVerticeOrigem	Identificador do vertice de origem.
idVerticeDestino	Identificador do vertice de destino.
peso	Peso da aresta.

```
14                                                                          {
15     this->idVerticeOrigem = idVerticeOrigem;
16     this->idVerticeDestino = idVerticeDestino;
17     this->peso = peso;
18     this->proximo = nullptr;
19 }
```

NoAresta::~NoAresta ()

Destrutor da classe **NoAresta**.

```
24                      {
25
26 }
```

Documentação das funções

int NoAresta::getDestino ()

Retorna o vertice de destino.

Retorna o vertice de destino da aresta.

Retorna

Identificador do vertice de destino.

```
41                           {
42     return idVerticeDestino;
43 }
```

int NoAresta::getOrigem ()

Retorna o vertice de origem.

Retorna o vertice de origem da aresta.

Retorna

Identificador do vertice de origem.

```
33                           {
34     return idVerticeOrigem;
35 }
```

float NoAresta::getPeso ()

Retorna o peso da aresta.

Retorna

Peso da aresta.

```
49                           {
```

```
50     return peso;
51 }
```

NoAresta * NoAresta::getProximo ()

Retorna o proximo no aresta.

Retorna o proximo Noh aresta.

Retorna

Ponteiro para o proximo no aresta.

Ponteiro para o proximo Noh aresta.

```
57                                  {
58     return proximo;
59 }
```

void NoAresta::setProximo (NoAresta * proximo)

Define o proximo no aresta.

Define o proximo Noh aresta.

Parâmetros

proximo	Ponteiro para o proximo no aresta.
proximo	Ponteiro para o proximo Noh aresta.

```
65                                              {
66     this->proximo = proximo;
67 }
```

void NoAresta::setVerticeDestino (int novoId)

Define o identificador do vertice de destino.

Define o id do vertice de destino.

Parâmetros

novoId	Novo identificador do vertice de destino.

```
81                                              {
82     this->idVerticeDestino = novoId;
83 }
```

void NoAresta::setVerticeOrigem (int novoId)

Define o identificador do vertice de origem.

Define o id do vertice de origem.

Parâmetros

novoId	Novo identificador do vertice de origem.

```
73                                              {
74     this->idVerticeOrigem = novoId;
75 }
```

A documentação para esta classe foi gerada a partir dos seguintes ficheiros:

- TrabalhoGrafos/include/**NoAresta.h**
- TrabalhoGrafos/src/**NoAresta.cpp**

- TrabalhoGrafos/src/**NoVertice.cpp**

Referência à classe NoVertice

Classe que representa um nó de vértice em uma lista de adjacência.
`#include <NoVertice.h>`

Membros públicos

- **NoVertice** (int vertice, float peso)
  *Construtor da classe **NoVertice**.*

- **~NoVertice** ()
  *Destrutor da classe **NoVertice**.*

- int **getIdVertice** ()
  Retorna o identificador do vertice.

- float **getPesoVertice** ()
  Retorna o peso do vertice.

- int **getNumArestas** ()
  Retorna a quantidade de arestas do vertice.

- **NoVertice * getProximo** ()
  Retorna o proximo no vertice.

- **ListaAdjAresta * getArestas** ()
  Retorna a lista encadeada de arestas.

- void **setProximo** (**NoVertice** *proximo)
  Define o proximo no vertice.

- int **setIdVertice** (int novoId)
  Define o identificador do vertice.

- int **getNumVizinhos** ()
  Retorna o numero de vizinhos do vertice (numero de arestas de saida/ grau do vertice).

- void **adicionar_aresta** (int id, float peso)
  Adiciona uma aresta ao vertice.

- void **remover_aresta** (int destino)
  Remove uma aresta do vertice.

- void **remover_primeira_aresta** ()
  Remove a primeira aresta do vertice.

Descrição detalhada

Classe que representa um nó de vértice em uma lista de adjacência.

Documentação dos Construtores & Destrutor

NoVertice::NoVertice (int vertice, float peso)

Construtor da classe **NoVertice**.

Parâmetros

vertice	Identificador do vertice.
peso	Peso do vertice.

NoVertice::~NoVertice ()

Destrutor da classe **NoVertice**.

Documentação das funções

void NoVertice::adicionar_aresta (int id, float peso)

Adiciona uma aresta ao vertice.

Parâmetros

id	Identificador do vertice de destino.
peso	Peso da aresta.

ListaAdjAresta * NoVertice::getArestas ()

Retorna a lista encadeada de arestas.

Retorna

Ponteiro para a lista de arestas.

int NoVertice::getIdVertice ()

Retorna o identificador do vertice.

Retorna

Identificador do vertice.

int NoVertice::getNumArestas ()

Retorna a quantidade de arestas do vertice.

Retorna

Numero de arestas do vertice.

int NoVertice::getNumVizinhos ()

Retorna o numero de vizinhos do vertice (numero de arestas de saida/ grau do vertice).

Retorna

Numero de vizinhos do vertice.

float NoVertice::getPesoVertice ()

Retorna o peso do vertice.

Retorna

Peso do vertice.

NoVertice * NoVertice::getProximo ()

Retorna o proximo no vertice.

Retorna

Ponteiro para o proximo no vertice.

void NoVertice::remover_aresta (int destino)

Remove uma aresta do vertice.

Parâmetros

destino	Identificador do vertice de destino.

void NoVertice::remover_primeira_aresta ()

Remove a primeira aresta do vertice.

int NoVertice::setIdVertice (int novoId)

Define o identificador do vertice.

Parâmetros

novoId	Novo identificador do vertice.

void NoVertice::setProximo (NoVertice * proximo)

Define o proximo no vertice.

Parâmetros

proximo	Ponteiro para o proximo no vertice.

A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- TrabalhoGrafos/include/**NoVertice.h**

Documentação do ficheiro

Referência ao ficheiro TrabalhoGrafos/include/Grafo.h

```
#include "../include/ListaAdjAresta.h"
#include "../include/ListaAdjVertice.h"
#include <iostream>
```

Componentes

class **Grafo***Classe base para representar um grafo.*

Grafo.h

Ir para a documentação deste ficheiro.

```
1 #ifndef GRAFO_H
2 #define GRAFO_H
3
4 #include "../include/ListaAdjAresta.h"
5 #include "../include/ListaAdjVertice.h"
6 #include <iostream>
7
8 using namespace std;
9
14 class Grafo {
15 protected:
16     int numVertices;
17     bool direcionado;
18     bool ponderadoVertices;
19     bool ponderadoArestas;
20
21 public:
29     Grafo(int numVertices, bool direcionado, bool ponderadoVertices, bool
ponderadoArestas);
30
34     ~Grafo();
35
40     int n_conexo();
41
46     int get_grau();
47
52     int get_ordem();
53
58     bool eh_direcionado();
59
64     bool vertice_ponderado();
65
70     bool aresta_ponderada();
71
76     bool eh_completo();
77
83     static void carrega_grafo(Grafo* grafo, const string& nomeArquivo);
84
88     void imprime();
89
90     // Funcoes auxiliares abstratas que serao implementadas nas classes filhas
91     virtual ListaAdjAresta* get_vizinhos(int id) { return nullptr; };
92     virtual int get_num_vizinhos(int id) { return 0; };
93     virtual void dfs(int v, bool* visitado){};
94     virtual bool existe_vertice(int id)=0;
95
96     // Funcoes de manipulacao de vertices e arestas abstratas que serao implementadas
nas classes filhas
97     virtual void adicionar_vertice(int id, float peso = 0.0){};
98     virtual void adicionar_aresta(int origem, int destino, float peso = 1.0){};
99     virtual void remover_primeira_aresta(int id){};
100     virtual void remover_vertice(int id){};
101     virtual void remover_aresta(int origem, int destino){};
102
103     virtual int calcula_menor_dist(int origem, int destino)=0;
104     virtual int calcula_maior_menor_dist()=0;
105 };
106
107 #endif
```

Referência ao ficheiro TrabalhoGrafos/include/GrafoLista.h

```
#include "Grafo.h"
#include "ListaAdjVertice.h"
#include "ListaAdjAresta.h"
```

Componentes

class **GrafoLista***Classe que representa um grafo utilizando lista de adjacência.*

GrafoLista.h

Ir para a documentação deste ficheiro.

```
1 #ifndef GRAFO_LISTA_H
2 #define GRAFO_LISTA_H
3
4 #include "Grafo.h"
5 #include "ListaAdjVertice.h"
6 #include "ListaAdjAresta.h"
7
12 class GrafoLista : public Grafo {
13 protected:
14     ListaAdjVertice* listaAdjVertices;
15
16 public:
24     GrafoLista(int numVertices, bool direcionado, bool ponderadoVertices, bool
ponderadoArestas);
25
29     ~GrafoLista();
30
36     int get_num_vizinhos(int id) override;
37
43     void dfs(int id, bool* visitado) override;
44
50     bool existe_vertice(int id) override;
51
57     void adicionar_vertice(int id, float peso = 0.0) override;
58
65     void adicionar_aresta(int origem, int destino, float peso = 1.0) override;
66
71     void remover_vertice(int id) override;
72
78     void remover_aresta(int origem, int destino) override;
79
84     void remover_primeira_aresta(int id) override;
85
92     int calcula_menor_dist(int origem, int destino);
93
98     int calcula_maior_menor_dist();
99
103     void imprimeGrafoLista();
104
108     void imprimeListaAdj();
109 };
110
111 #endif // GRAFO_LISTA_H
```

Referência ao ficheiro TrabalhoGrafos/include/GrafoMatriz.h

```
#include "Grafo.h"
```

Componentes

class **GrafoMatriz***Classe que representa um grafo utilizando matriz de adjacência.*

GrafoMatriz.h

Ir para a documentação deste ficheiro.

```
1 #ifndef GRAFOMATRIZ_H
2 #define GRAFOMATRIZ_H
3
4 #include "Grafo.h"
5
10 class GrafoMatriz : public Grafo {
11 protected:
12     int** matrizAdj;
13     int tamanhoMatriz;
14
15 public:
23     GrafoMatriz(int numVertices, bool direcionado, bool ponderadoVertices, bool
ponderadoArestas);
24
28     ~GrafoMatriz();
29
35     int grauVertice(int vertice);
36
42     int get_num_vizinhos(int id) override;
43
49     void dfs(int id, bool* visitado) override;
50
56     bool existe_vertice(int id) override;
57
63     void adicionar_vertice(int id, float peso = 0.0) override;
64
71     void adicionar_aresta(int origem, int destino, float peso = 1.0) override;
72
77     void remover_vertice(int id) override;
78
84     void remover_aresta(int origem, int destino) override;
85
90     void remover_primeira_aresta(int id) override;
91
98     int calcula_menor_dist(int origem, int destino) override;
99
104     int calcula_maior_menor_dist() override;
105
109     void imprimirMatrizAdj();
110
114     void imprimeGrafoMatriz();
115 };
116
117 #endif // GRAFOMATRIZ_H
```

Referência ao ficheiro TrabalhoGrafos/include/ListaAdjAresta.h

```
#include "NoAresta.h"
```

Componentes

class **ListaAdjAresta***Classe que representa uma lista de adjacência de arestas.*

ListaAdjAresta.h

Ir para a documentação deste ficheiro.

```
1 #ifndef LISTA_ADJ_ARESTA_H
2 #define LISTA_ADJ_ARESTA_H
3
4 #include "NoAresta.h"
5
10 class ListaAdjAresta {
11 private:
12     NoAresta* cabeca;
13
14 public:
18     ListaAdjAresta();
19
23     ~ListaAdjAresta();
24
29     NoAresta* getCabeca();
30
35     int getNumVerticesVizinhos();
36
43     void adicionar_aresta(int origem, int destino, float peso);
44
50     void remover_aresta(int origem, int destino);
51
55     void remover_primeira_aresta();
56 };
57
58 #endif // LISTA_ADJ_ARESTA_H
```

Referência ao ficheiro TrabalhoGrafos/include/ListaAdjVertice.h

```
#include "NoAresta.h"
```

Componentes

class **ListaAdjAresta***Classe que representa uma lista de adjacência de arestas.*

ListaAdjVertice.h

Ir para a documentação deste ficheiro.

```
1 #ifndef LISTA_ADJ_ARESTA_H
2 #define LISTA_ADJ_ARESTA_H
3
4 #include "NoAresta.h"
5
10 class ListaAdjAresta {
11 private:
12     NoAresta* cabeca;
13
14 public:
18     ListaAdjAresta();
19
23     ~ListaAdjAresta();
24
29     NoAresta* getCabeca();
30
35     int getNumVerticesVizinhos();
36
43     void adicionar_aresta(int origem, int destino, float peso);
44
50     void remover_aresta(int origem, int destino);
51
55     void remover_primeira_aresta();
56 };
57
58 #endif // LISTA_ADJ_ARESTA_H
```

Referência ao ficheiro TrabalhoGrafos/include/NoAresta.h

Componentes

class **NoAresta***Classe que representa um nó de aresta em uma lista de adjacência.*

NoAresta.h

Ir para a documentação deste ficheiro.

```
1 #ifndef NO_ARESTA_H
2 #define NO_ARESTA_H
3
8 class NoAresta {
9 private:
10     int idVerticeOrigem;
11     int idVerticeDestino;
12     float peso;
13     NoAresta* proximo;
14
15 public:
22     NoAresta(int idVerticeOrigem, int idVerticeDestino, float peso);
23
27     ~NoAresta();
28
33     int getOrigem();
34
39     int getDestino();
40
45     float getPeso();
46
51     NoAresta* getProximo();
52
57     void setProximo(NoAresta* proximo);
58
63     void setVerticeOrigem(int novoId);
64
69     void setVerticeDestino(int novoId);
70 };
71
72 #endif // NO_ARESTA_H
```

Referência ao ficheiro TrabalhoGrafos/include/NoVertice.h

```
#include "ListaAdjAresta.h"
#include "NoAresta.h"
```

Componentes

class **NoVertice***Classe que representa um nó de vértice em uma lista de adjacência.*

NoVertice.h

Ir para a documentação deste ficheiro.

```
1 #ifndef NO_VERTICE_H
2 #define NO_VERTICE_H
3
4 #include "ListaAdjAresta.h"
5 #include "NoAresta.h"
6
11 class NoVertice {
12 private:
13     int idVertice;
14     float peso;
15     int numArestas;
16     NoVertice* proximo;
17     ListaAdjAresta* arestas;
18
19 public:
25     NoVertice(int vertice, float peso);
26
30     ~NoVertice();
31
36     int getIdVertice();
37
42     float getPesoVertice();
43
48     int getNumArestas();
49
54     NoVertice* getProximo();
55
60     ListaAdjAresta* getArestas();
61
66     void setProximo(NoVertice* proximo);
67
72     int setIdVertice(int novoId);
73
78     int getNumVizinhos();
79
85     void adicionar_aresta(int id, float peso);
86
91     void remover_aresta(int destino);
92
96     void remover_primeira_aresta();
97 };
98
99 #endif // NO_VERTICE_H
```

Referência ao ficheiro TrabalhoGrafos/main.cpp

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

Funções

- int **main** (int argc, char *argv[])
  Funcao principal do programa.

Documentação das funções

int main (int argc, char * argv[])

Funcao principal do programa.

Para compilar: CTRL+SHIFT+B ou g++ -o main **main.cpp src/Grafo.cpp src/GrafoMatriz.cpp src/GrafoLista.cpp src/ListaAdjAresta.cpp src/ListaAdjVertice.cpp src/NoAresta.cpp src/NoVertice.cpp** Para executar Matriz: ./main -d -m grafo.txt Para executar Lista: ./main -d -l grafo.txt

Parâmetros

argc	Numero de argumentos.
argv	Vetor de argumentos.

Retorna

int Codigo de retorno.

```
22                                  {
23     string modo = argv[2];
24     string arquivo = "./entradas/" + string(argv[3]);
25 
26     if (modo == "-m") {
27         cout << "============================== MATRIZ
==============================" << endl;
28         GrafoMatriz grafoMatriz(0, false, false, false);
29         grafoMatriz.carrega_grafo(&grafoMatriz, arquivo);
30         grafoMatriz.imprimeGrafoMatriz();
31         cout << "============================== FIM MATRIZ
==============================" << endl << endl;
32     } else if (modo == "-l") {
33         cout << "============================== LISTA
==============================" << endl;
34         GrafoLista grafoLista(0, false, false, false);
35         grafoLista.carrega_grafo(&grafoLista, arquivo);
36         grafoLista.imprimeGrafoLista();
37         cout << "============================== FIM LISTA
==============================" << endl << endl;
38     } else {
39         cerr << "Use ./main -d -m grafo.txt  ou ./main -d -l grafo.txt" << endl;
40         return 1;
41     }
42 
43     return 0;
44 }
```

Referência ao ficheiro TrabalhoGrafos/src/Grafo.cpp

```
#include "../include/Grafo.h"
#include <iostream>
#include <fstream>
#include <cmath>
#include <string>
```

Referência ao ficheiro TrabalhoGrafos/src/GrafoLista.cpp

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

Referência ao ficheiro TrabalhoGrafos/src/GrafoMatriz.cpp

```
#include "../include/GrafoMatriz.h"
#include <fstream>
#include <iostream>
#include <cmath>
#include <string>
#include <iomanip>
```

Referência ao ficheiro TrabalhoGrafos/src/ListaAdjAresta.cpp

```
#include "../include/ListaAdjAresta.h"
#include "../include/NoAresta.h"
#include <iostream>
```

Referência ao ficheiro TrabalhoGrafos/src/ListaAdjVertice.cpp

```
#include "../include/ListaAdjVertice.h"
#include "../include/NoVertice.h"
#include <iostream>
```

Referência ao ficheiro TrabalhoGrafos/src/NoAresta.cpp

```
#include "../include/NoAresta.h"
#include <iostream>
```

Referência ao ficheiro TrabalhoGrafos/src/NoVertice.cpp

```
#include "../include/NoAresta.h"
#include <iostream>
```