Trabalho de Teoria dos Grafos

Documentação – Parte 3

Índice Hierárquico

Hierarquia de Classes

Esta lista de hierarquias está parcialmente ordenada (ordem alfabética):

Índice dos Componentes

Lista de Classes

Aqui estão as classes, estruturas, uniões e interfaces e suas respectivas descrições:

Classes

Referência da Classe Grafo

Representa um grafo com propriedades básicas.

`#include <Grafo.h>`

Diagrama de hierarquia da classe Grafo:

IMAGE

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 (número de vértices).

- bool **eh_direcionado** ()

Verifica se o grafo é direcionado.

- bool **vertice_ponderado** ()

Verifica se o grafo possui peso nos vértices.

- bool **aresta_ponderada** ()

Verifica se o grafo possui peso nas arestas.

- bool **eh_completo** ()

Verifica se o grafo é completo.

- void **imprimir_descricao** ()

Imprime os atributos básicos do grafo.

- void **imprimir_algoritmos_cobertura_vertice** (**Grafo** *grafo)

Imprime os resultados dos algoritmos gulosos para cobertura de vértices.

- void **analise_algoritmos_cobertura_vertice** (**Grafo** *grafo, int numVezes)

Analisa os algoritmos gulosos para cobertura de vértices.

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

Retorna os vértices vizinhos de um vértice.

- virtual int **get_num_vizinhos** (int id)

Retorna o número de vértices vizinhos de um vértice.

- virtual void **dfs** (int v, bool *visitado)

Realiza a busca em profundidade (DFS) a partir de um vértice.

- virtual bool **existe_vertice** (int id)=0

Verifica se um vértice existe no grafo.

- int **get_num_arestas_grafo** ()

Retorna o número de arestas do grafo.

- void **incrementa_num_vertices_grafo** ()

Incrementa o número de vértices do grafo.

- void **decrementa_num_vertices_grafo** ()

Decrementa o número de vértices do grafo.

- void **incrementa_num_arestas_grafos** ()

Incrementa o número de arestas do grafo.

- void **decrementa_num_arestas_grafos** ()

Decrementa o número de arestas do grafo.

- void **diminui_num_arestas_grafos** (int valor)

Diminui o número de arestas do grafo por um valor especificado.

- virtual void **adicionar_vertice** (int id, float peso=0.0)

Adiciona um vértice ao grafo.

- virtual void **adicionar_aresta** (int origem, int destino, float peso=1.0)

Adiciona uma aresta ao grafo.

- virtual void **remover_primeira_aresta** (int id)

Remove a primeira aresta associada a um vértice.

- virtual void **remover_vertice** (int id)

Remove um vértice do grafo.

- virtual void **remover_aresta** (int origem, int destino)

Remove uma aresta do grafo.

- virtual int **calcula_menor_dist** (int origem, int destino)=0

Calcula a menor distância entre dois vértices.

- virtual int **calcula_maior_menor_dist** ()

Calcula a maior dentre as menores distâncias entre pares de vértices.

- virtual int **alg_guloso_cobertura_vertice** ()=0

Executa o algoritmo guloso para cobertura de vértices.

- virtual int **alg_randomizado_cobertura_vertice** ()=0

Executa o algoritmo randomizado para cobertura de vértices.

- virtual int **alg_reativo_cobertura_vertice** ()=0

Executa o algoritmo reativo para cobertura de vértices.

Membros públicos estáticos

- static void **carrega_grafo** (**Grafo** *grafo, const string &nomeArquivo)

Carrega um grafo a partir de um arquivo.

Atributos Protegidos

- int **numVertices**

Número de vértices (ordem) do grafo.

- int **numArestasGrafo**

Número de arestas do grafo.

- bool **direcionado**

Indica se o grafo é direcionado.

- bool **ponderadoVertices**

Indica se o grafo possui peso nos vértices.

- bool **ponderadoArestas**

Indica se o grafo possui peso nas arestas.

Descrição detalhada

Representa um grafo com propriedades básicas.

A classe **Grafo** contém os atributos e métodos essenciais para definir um grafo, como o número de vértices, número de arestas, e flags que indicam se o grafo é direcionado ou se possui pesos em seus vértices e arestas. Métodos abstratos para manipulação e análise de grafos devem ser implementados pelas classes derivadas.

Construtores e Destrutores

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

Construtor da classe **Grafo**.

Parâmetros

numVertices	Número de vértices do grafo.
direcionado	Indica se o grafo é direcionado.
ponderadoVertices	Indica se os vértices possuem peso.
ponderadoArestas	Indica se as arestas possuem peso.

```
14 {
15      this->numVertices = numVertices;
16      this->direcionado = direcionado;
17      this->ponderadoVertices = ponderadoVertices;
18      this->ponderadoArestas = ponderadoArestas;
19      this->numArestasGrafo = 0;
20 }
```

Grafo::~Grafo ()

Destrutor da classe **Grafo**.

```
23 {
24 }
```

Documentação das funções

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

Adiciona uma aresta ao grafo.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.
peso	Peso da aresta (valor padrão 1.0).

Reimplementado por **GrafoLista** (*p.22*) e **GrafoMatriz** (*p.38*).

```
198 {};
```

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

Adiciona um vértice ao grafo.

Parâmetros

id	Identificador do vértice.
peso	Peso do vértice (valor padrão 0.0).

Reimplementado por **GrafoLista** (*p.22*) e **GrafoMatriz** (*p.39*).

```
190 {};
```

virtual int Grafo::alg_guloso_cobertura_vertice () `[pure virtual]`

Executa o algoritmo guloso para cobertura de vértices.

Retorna

Resultado do algoritmo (por exemplo, tamanho da cobertura).

Implementado por **GrafoLista** (*p.23*) e **GrafoMatriz** (*p.40*).

virtual int Grafo::alg_randomizado_cobertura_vertice () `[pure virtual]`

Executa o algoritmo randomizado para cobertura de vértices.

Retorna

Resultado do algoritmo.

Implementado por **GrafoLista** (*p.25*) e **GrafoMatriz** (*p.42*).

virtual int Grafo::alg_reativo_cobertura_vertice () `[pure virtual]`

Executa o algoritmo reativo para cobertura de vértices.

Retorna

Resultado do algoritmo.

Implementado por **GrafoLista** (*p.26*) e **GrafoMatriz** (*p.46*).

void Grafo::analise_algoritmos_cobertura_vertice (Grafo * grafo, int numVezes)

Analisa os algoritmos gulosos para cobertura de vértices.

Parâmetros

grafo	Ponteiro para o objeto **Grafo**.

numVezes	Número de vezes a serem executados os algoritmos.

```
153 {
154     double tempoGulosoTotal = 0.0, tempoRandomizadoTotal = 0.0, tempoReativoTotal
= 0.0;
155      int verticesSolucaoGuloso[numVezes], verticesSolucaoRandomizado[numVezes],
verticesSolucaoReativo[numVezes];
156
157       clock_t startFor = clock();
158       for(int i = 0; i < numVezes; i++) {
159           cout << endl << "-----------------------------------" << "Execucao " <<
i+1 << "-----------------------------------" << endl;
160
161            // Mede o tempo do algoritmo guloso
162            clock t start = clock();
163            verticesSolucaoGuloso[i] = grafo->alg_guloso_cobertura_vertice();
164            clock_t end = clock();
165            double tempoGuloso = double(end - start) / CLOCKS PER SEC;
166            tempoGulosoTotal += tempoGuloso;
167
168            // Mede o tempo do algoritmo randomizado
169            start = clock();
170            verticesSolucaoRandomizado[i] =
grafo->alg_randomizado_cobertura_vertice();
171            end = clock();
172            double tempoRandomizado = double(end - start) / CLOCKS_PER_SEC;
173            tempoRandomizadoTotal += tempoRandomizado;
174
175            // Mede o tempo do algoritmo reativo
176            start = clock();
177            verticesSolucaoReativo[i] = grafo->alg_reativo_cobertura_vertice();
178            end = clock();
179            double tempoReativo = double(end - start) / CLOCKS PER SEC;
180            tempoReativoTotal += tempoReativo;
181       }
182       clock t endFor = clock();
183       double durationFor = double(endFor - startFor) / CLOCKS_PER_SEC;
184
185       int melhorGuloso = verticesSolucaoGuloso[0], melhorRandomizado =
verticesSolucaoRandomizado[0], melhorReativo = verticesSolucaoReativo[0];
186      int indiceMelhorGuloso = 0, indiceMelhorRandomizado = 0, indiceMelhorReativo
= 0;
187      int somaVerticesGuloso = 0, somaVerticesRandomizado = 0, somaVerticesReativo
= 0;
188
189       for(int i = 0; i < numVezes; i++) {
190            if(verticesSolucaoGuloso[i] < melhorGuloso) {
191                 melhorGuloso = verticesSolucaoGuloso[i];
192                 indiceMelhorGuloso = i;
193            }
194            if(verticesSolucaoRandomizado[i] < melhorRandomizado) {
195                 melhorRandomizado = verticesSolucaoRandomizado[i];
196                 indiceMelhorRandomizado = i;
197            }
198            if(verticesSolucaoReativo[i] < melhorReativo) {
199                 melhorReativo = verticesSolucaoReativo[i];
200                 indiceMelhorReativo = i;
201            }
202            somaVerticesGuloso += verticesSolucaoGuloso[i];
203            somaVerticesRandomizado += verticesSolucaoRandomizado[i];
204            somaVerticesReativo += verticesSolucaoReativo[i];
205       }
206
207       cout << endl <<
"-----------------------------------------Analises--------------------------------
--------------" << endl;
208     cout << "Tempo de execucao Total das " << numVezes << " execucoes: " << durationFor
<< " segundos" << endl;
209
```

```
210       cout << left << setw(40) << "Tempo Total Algoritmo Guloso:"
211             << setw(10) << tempoGulosoTotal << " segundos | "
212           << "Media: " << setw(10) << (tempoGulosoTotal / numVezes) << " segundos
| "
213           << "Media de vertices: " << setw(10) << (somaVerticesGuloso / numVezes)
<< " | "
214             << "Melhor Solucao: " << setw(5) << melhorGuloso << " | "
215             << "Execucao: " << indiceMelhorGuloso << endl;
216
217       cout << left << setw(40) << "Tempo Total Algoritmo Randomizado:"
218             << setw(10) << tempoRandomizadoTotal << " segundos | "
219             << "Media: " << setw(10) << (tempoRandomizadoTotal / numVezes) << "
segundos | "
220         << "Media de vertices: " << setw(10) << (somaVerticesRandomizado / numVezes)
<< " | "
221             << "Melhor Solucao: " << setw(5) << melhorRandomizado << " | "
222             << "Execucao: " << indiceMelhorRandomizado << endl;
223
224       cout << left << setw(40) << "Tempo Total Algoritmo Reativo:"
225             << setw(10) << tempoReativoTotal << " segundos | "
226           << "Media: " << setw(10) << (tempoReativoTotal / numVezes) << " segundos
| "
227           << "Media de vertices: " << setw(10) << (somaVerticesReativo / numVezes)
<< " | "
228             << "Melhor Solucao: " << setw(5) << melhorReativo << " | "
229             << "Execucao: " << indiceMelhorReativo << endl;
230 }
```

bool Grafo::aresta_ponderada ()

Verifica se o grafo possui peso nas arestas.

Retorna

True se as arestas são ponderadas, caso contrário, false.

```
79 {
80       return ponderadoArestas;
81 }
```

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

Calcula a maior dentre as menores distâncias entre pares de vértices.

Retorna

A maior das menores distâncias encontradas.

```
272                                                 {
273       int maiorMenorDist = 0;
274       int verticeOrigem = -1;
275       int verticeDestino = -1;
276
277       for (int i = 1; i <= numVertices; i++) {
278            for (int j = 1; j <= numVertices; j++) {
279                 if (i != j) {
280                      int menorDist = calcula_menor_dist(i, j);
281                      if (menorDist != 1000000 && menorDist > maiorMenorDist) {
282                           maiorMenorDist = menorDist;
283                           verticeOrigem = i;
```

```
284                             verticeDestino = j;
285                         }
286                     }
287                 }
288         }
289
290         if (verticeOrigem != -1 && verticeDestino != -1) {
291           cout << "Maior menor distancia: (" << verticeOrigem << "-" << verticeDestino
<< ") = " << maiorMenorDist << endl;
292         } else {
293             cout << "Nao ha caminhos validos no grafo." << endl;
294         }
295
296         return maiorMenorDist;
297 }
```

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

Calcula a menor distância entre dois vértices.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

Retorna

Menor distância entre os vértices.

Implementado por **GrafoLista** (*p.29*) e **GrafoMatriz** (*p.48*).

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

Carrega um grafo a partir de um arquivo.

Parâmetros

grafo	Ponteiro para o objeto **Grafo**.
nomeArquivo	Nome do arquivo que contém a descrição do grafo.

```
234
  {
235         ifstream arquivo(nomeArquivo);
236         if (!arquivo.is_open()) {
237             cerr << "Erro ao abrir o arquivo: " << nomeArquivo << endl;
238             return;
239         }
240         int numVertices, direcionado, ponderadoVertices, ponderadoArestas;
241       arquivo >> numVertices >> direcionado >> ponderadoVertices >> ponderadoArestas;
242
243         grafo->direcionado = direcionado;
244         grafo->ponderadoVertices = ponderadoVertices;
245         grafo->ponderadoArestas = ponderadoArestas;
246
247         for(int i = 1; i <= numVertices; i++) {
248             if(ponderadoVertices == 1) {
249                 float peso;
250                 arquivo >> peso;
251                 grafo->adicionar_vertice(i, peso);
252             } else {
253                 grafo->adicionar_vertice(i);
254             }
```

```
255
256       }
257
258       int origem, destino;
259       while (arquivo >> origem >> destino) {
260            if (ponderadoArestas == 1) {
261                 float peso;
262                 arquivo >> peso;
263                 grafo->adicionar_aresta(origem, destino, peso);
264            } else {
265                 grafo->adicionar aresta(origem, destino);
266            }
267       }
268       arquivo.close();
269 }
```

void Grafo::decrementa_num_arestas_grafos ()

Decrementa o número de arestas do grafo.

```
109 {
110       numArestasGrafo--;
111 }
```

void Grafo::decrementa_num_vertices_grafo ()

Decrementa o número de vértices do grafo.

```
127 {
128       numVertices--;
129 }
```

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

Realiza a busca em profundidade (DFS) a partir de um vértice.

Parâmetros

v	Vértice inicial.
visitado	Vetor indicando os vértices já visitados.

Reimplementado por **GrafoLista** (*p.30*) e **GrafoMatriz** (*p.49*).

```
142 {};
```

void Grafo::diminui_num_arestas_grafos (int valor)

Diminui o número de arestas do grafo por um valor especificado.

Parâmetros

valor	Valor a ser subtraído do total de arestas.

```
115 {
116       numArestasGrafo = numArestasGrafo - valor;
117 }
```

bool Grafo::eh_completo ()

Verifica se o grafo é completo.

Retorna

True se o grafo for completo, caso contrário, false.

```
85 {
86       for (int i = 1; i <= numVertices; i++) {
87            int numVizinhos = get num vizinhos(i);
88            if (numVizinhos != numVertices - 1) {
89                 return false;
90            }
91       }
92       return true;
93 }
```

bool Grafo::eh_direcionado ()

Verifica se o grafo é direcionado.

Retorna

True se for direcionado, caso contrário, false.

```
67 {
68       return direcionado;
69 }
```

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

Verifica se um vértice existe no grafo.

Parâmetros

id	Identificador do vértice.

Retorna

True se o vértice existir, caso contrário, false.

Implementado por **GrafoLista** (*p.30*) e **GrafoMatriz** (*p.50*).

int Grafo::get_grau ()

Retorna o grau do grafo.

Retorna

Grau do grafo.

```
49 {
50       int grauMax = 0;
51       for (int i = 1 ; i <= numVertices; i++) {
52            if (get_num_vizinhos(i) > grauMax && existe_vertice(i)) {
53                 grauMax = get_num_vizinhos(i);
54            }
55       }
56       return grauMax;
57 }
```

int Grafo::get_num_arestas_grafo ()

Retorna o número de arestas do grafo.

Retorna

Número de arestas.

```
97 {
98       return numArestasGrafo;
99 }
```

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

Retorna o número de vértices vizinhos de um vértice.

Parâmetros

id	Identificador do vértice.

Retorna

Número de vizinhos.

Reimplementado por **GrafoLista** (*p.31*) e **GrafoMatriz** (*p.50*).

```
135 { return 0; };
```

int Grafo::get_ordem ()

Retorna a ordem do grafo (número de vértices).

Retorna

Ordem do grafo.

```
61 {
62       return numVertices;
63 }
```

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

Retorna os vértices vizinhos de um vértice.

Parâmetros

id	Identificador do vértice.

Retorna

Ponteiro para a lista de adjacência de arestas dos vizinhos.

```
128 { return nullptr; };
```

void Grafo::imprimir_algoritmos_cobertura_vertice (Grafo * grafo)

Imprime os resultados dos algoritmos gulosos para cobertura de vértices.

Parâmetros

grafo	Ponteiro para o objeto **Grafo**.

```
146 {
147       grafo->alg_guloso_cobertura_vertice();
148       grafo->alg randomizado cobertura vertice();
149       grafo->alg_reativo_cobertura_vertice();
150 }
```

void Grafo::imprimir_descricao ()

Imprime os atributos básicos do grafo.

```
133 {
134       cout << "*** Descricao do Grafo ***" << endl;
135       cout << "Grau: " << get_grau() << endl;
136       cout << "Ordem: " << numVertices << endl;
137       cout << "Direcionado: " << (eh_direcionado() ? "Sim" : "Nao") << endl;
138       cout << "Componentes conexas: " << n_conexo() << endl;
139      cout << "Vertices ponderados: " << (vertice_ponderado() ? "Sim" : "Nao") <<
endl;
140     cout << "Arestas ponderadas: " << (aresta_ponderada() ? "Sim" : "Nao") << endl;
141       cout << "Completo: " << (eh_completo() ? "Sim" : "Nao") << endl;
142 }
```

void Grafo::incrementa_num_arestas_grafos ()

Incrementa o número de arestas do grafo.

```
103 {
104       numArestasGrafo++;
105 }
```

void Grafo::incrementa_num_vertices_grafo ()

Incrementa o número de vértices do grafo.

```
121 {
122       numVertices++;
123 }
```

int Grafo::n_conexo ()

Retorna a quantidade total de componentes conexas do grafo.

Retorna

Número de componentes conexas.

```
29 {
30       bool visitado[numVertices];
31
32       for (int i = 1; i <= numVertices; i++) {
33            visitado[i] = false;
34       }
35
36       int numComponentes = 0;
37
38       for (int i = 1; i <= numVertices; i++) {
39            if (!visitado[i] && this->existe_vertice(i)) {
40                 dfs(i, visitado);
41                 numComponentes++;
42            }
43       }
44       return numComponentes;
45 }
```

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

Remove uma aresta do grafo.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

Reimplementado por **GrafoLista** (*p.31*) e **GrafoMatriz** (*p.51*).

```
217 {};
```

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

Remove a primeira aresta associada a um vértice.

Parâmetros

id	Identificador do vértice.

Reimplementado por **GrafoLista** (*p.32*) e **GrafoMatriz** (*p.52*).

```
204 {};
```

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

Remove um vértice do grafo.

Parâmetros

id	Identificador do vértice a ser removido.

Reimplementado por **GrafoLista** (*p.32*) e **GrafoMatriz** (*p.52*).

```
210 {};
```

bool Grafo::vertice_ponderado ()

Verifica se o grafo possui peso nos vértices.

Retorna

True se os vértices são ponderados, caso contrário, false.

```
73 {
74       return ponderadoVertices;
75 }
```

Atributos

bool Grafo::direcionado [protected]

Indica se o grafo é direcionado.

int Grafo::numArestasGrafo [protected]

Número de arestas do grafo.

int Grafo::numVertices [protected]

Número de vértices (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 vértices.

Referência da Classe GrafoLista

Representa um grafo utilizando listas de adjacência.

`#include <GrafoLista.h>`

Diagrama de hierarquia da classe GrafoLista:

IMAGE

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 número de vizinhos de um vértice.

- void **dfs** (int id, bool *visitado) override

Executa a busca em profundidade (DFS) a partir de um vértice.

- bool **existe_vertice** (int id) override

Verifica se um vértice existe no grafo.

- void **adicionar_vertice** (int id, float peso=0.0) override

Adiciona um vértice ao grafo.

- void **adicionar_aresta** (int origem, int destino, float peso=1.0) override

Adiciona uma aresta ao grafo.

- void **remover_vertice** (int id) override

Remove um vértice 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 associada a um vértice.

- int **calcula_menor_dist** (int origem, int destino)

Calcula a menor distância entre dois vértices.

- void **imprimeGrafoLista** ()

Imprime as informações do grafo.

- void **imprimeListaAdj** ()

Imprime a lista de adjacência do grafo.

- int **alg_guloso_cobertura_vertice** () override

Executa o algoritmo guloso para cobertura de vértices.

- int **alg_randomizado_cobertura_vertice** () override

Executa o algoritmo randomizado para cobertura de vértices.

- int **alg_reativo_cobertura_vertice** () override

Executa o algoritmo reativo para cobertura de vértices.

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 (número de vértices).

- bool **eh_direcionado** ()

Verifica se o grafo é direcionado.

- bool **vertice_ponderado** ()

Verifica se o grafo possui peso nos vértices.

- bool **aresta_ponderada** ()

Verifica se o grafo possui peso nas arestas.

- bool **eh_completo** ()

Verifica se o grafo é completo.

- void **imprimir_descricao** ()

Imprime os atributos básicos do grafo.

- void **imprimir_algoritmos_cobertura_vertice** (**Grafo** *grafo)

Imprime os resultados dos algoritmos gulosos para cobertura de vértices.

- void **analise_algoritmos_cobertura_vertice** (**Grafo** *grafo, int numVezes)

Analisa os algoritmos gulosos para cobertura de vértices.

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

Retorna os vértices vizinhos de um vértice.

- int **get_num_arestas_grafo** ()

Retorna o número de arestas do grafo.

- void **incrementa_num_vertices_grafo** ()

Incrementa o número de vértices do grafo.

- void **decrementa_num_vertices_grafo** ()

Decrementa o número de vértices do grafo.

- void **incrementa_num_arestas_grafos** ()

Incrementa o número de arestas do grafo.

- void **decrementa_num_arestas_grafos** ()

Decrementa o número de arestas do grafo.

- void **diminui_num_arestas_grafos** (int valor)

Diminui o número de arestas do grafo por um valor especificado.

- virtual int **calcula_maior_menor_dist** ()

Calcula a maior dentre as menores distâncias entre pares de vértices.

Atributos Protegidos

- **ListaAdjVertice** * **listaAdjVertices**

Lista encadeada de vértices.

Atributos Protegidos herdados de Grafo

- int **numVertices**

Número de vértices (ordem) do grafo.

- int **numArestasGrafo**

Número de arestas do grafo.

- bool **direcionado**

Indica se o grafo é direcionado.

- bool **ponderadoVertices**

Indica se o grafo possui peso nos vértices.

- 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)

Carrega um grafo a partir de um arquivo.

Descrição detalhada

Representa um grafo utilizando listas de adjacência.

A classe **GrafoLista** deriva de **Grafo** e implementa as operações de manipulação e análise de grafos utilizando uma estrutura de listas encadeadas para armazenar os vértices e suas respectivas arestas.

Construtores e Destrutores

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

Construtor da classe **GrafoLista**.

Parâmetros

numVertices	Número inicial de vértices.
direcionado	Indica se o grafo é direcionado.
ponderadoVertices	Indica se os vértices possuem peso.
ponderadoArestas	Indica se as arestas possuem peso.

```
11                                                  : Grafo(numVertices, direcionado,
ponderadoVertices, ponderadoArestas)
12 {
13       // Inicializa a lista de adjacencia
14       this->listaAdjVertices = new ListaAdjVertice(this);
15 }
```

GrafoLista::~GrafoLista ()

Destrutor da classe **GrafoLista**.

```
18 {
19       // Libera a memoria alocada para os vertices
20       delete this->listaAdjVertices;
21 }
```

Documentação das funções

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

Adiciona uma aresta ao grafo.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.
peso	Peso da aresta (valor padrão 1.0).

Reimplementa **Grafo** (*p.6*).

```
42
        {
43       // Verifica se o vertice de origem existe
44       if(!existe vertice(origem)){
45            cout << "Erro: Vertice " << origem << " nao existe!" << endl;
                                  /* { DEBUG } */
46            return;
47       }
48       // Verifica se o vertice de destino existe
49       if(!existe vertice(destino)){
50            cout << "Erro: Vertice " << destino << " nao existe!" << endl;
                                  /* { DEBUG } */
51            return;
52       }
53       // Verifica se a aresta eh um self-loop
54       if(origem == destino){
55          cout << "Erro: Nao eh possivel adicionar aresta. Origem e destino iguais!
" << endl;                /* { DEBUG } */
56            return;
57       }
58
59       // Adiciona a aresta
60       this->listaAdjVertices->adicionar_aresta(origem, destino, peso);
61       if(!direcionado){
62            this->listaAdjVertices->adicionar_aresta(destino, origem, peso);
63       }
64  }
```

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

Adiciona um vértice ao grafo.

Parâmetros

id	Identificador do novo vértice.
peso	Peso do vértice (valor padrão 0.0).

Reimplementa **Grafo** (*p.7*).

```
30                                                                    {
31       // Verifica se o vertice ja existe
32       if(existe_vertice(id)){
33            cout << "Erro: Vertice " << id << " ja existe!" << endl;
                                  /* { DEBUG } */
34            return;
```

```
35       }
36
37       // Adiciona o vertice
38       this->listaAdjVertices->adicionar_vertice(id, peso);
39  }
```

int GrafoLista::alg_guloso_cobertura_vertice () `[override]`, `[virtual]`

Executa o algoritmo guloso para cobertura de vértices.

Retorna

Resultado do algoritmo (por exemplo, tamanho da cobertura).

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

```
191                                                                 {
192       // Inicializa o tempo de execução
193       clock_t start = clock();
194
195       // Aloca e inicializa estruturas
196       bool* verticeEscolhido = new bool[numVertices + 1]();
197       bool* arestaCoberta = new bool[numArestasGrafo + 1]();
198       int* graus = new int[numVertices + 1]();
199       int* somaVizinhos = new int[numVertices + 1]();
200
201       for (int i = 1; i <= numVertices; i++) {
202            graus[i] = listaAdjVertices->getVertice(i)->getNumVizinhos();
203            NoAresta* arestaAtual =
listaAdjVertices->getVertice(i)->getArestas()->getCabeca();
204            while (arestaAtual != nullptr) {
205                 somaVizinhos[i] +=
listaAdjVertices->getVertice(arestaAtual->getDestino())->getNumVizinhos();
206                 arestaAtual = arestaAtual->getProximo();
207            }
208       }
209
210       int arestasCobertas = 0;
211       int qtdVerticesSolucao = 0;
212
213       // Algoritmo guloso
214       while (arestasCobertas < numArestasGrafo) {
215            int melhorVertice = -1;
216            int melhorSoma = -1;
217            int melhorGrau = -1;
218
219            for (int i = 1; i <= numVertices; i++) {
220                 if (!verticeEscolhido[i]) {
221                      int somaAtual = somaVizinhos[i];
222                      int grauAtual = graus[i];
223                      if ((somaAtual > melhorSoma) ||
224                           (somaAtual == melhorSoma && grauAtual > melhorGrau) ||
225                           (somaAtual == melhorSoma && grauAtual == melhorGrau &&
(melhorVertice == -1 || i < melhorVertice))) {
226                           melhorSoma = somaAtual;
227                           melhorGrau = grauAtual;
228                           melhorVertice = i;
229                      }
230                 }
231            }
232
233            // Se nao ha mais vertices para escolher, termina o algoritmo
234            if (melhorVertice == -1) break;
235
236            // Marca o vertice escolhido e atualiza as estruturas
```

```
237           verticeEscolhido[melhorVertice] = true;
238           qtdVerticesSolucao++;
239
240           // Atualiza os graus e as arestas cobertas
241           NoAresta* arestaAtual =
listaAdjVertices->getVertice(melhorVertice)->getArestas()->getCabeca();
242           while (arestaAtual != nullptr) {
243                int destino = arestaAtual->getDestino();
244                int idAresta = arestaAtual->getIdAresta();
245                if (!arestaCoberta[idAresta]) {
246                     arestaCoberta[idAresta] = true;
247                     arestasCobertas++;
248                     graus[destino]--;
249                     somaVizinhos[destino] -= graus[melhorVertice];
250                   // Se o grafo nao for direcionado, marca a aresta reversa como
coberta
251                     if (!direcionado) {
252                          NoAresta* reversa =
listaAdjVertices->getVertice(destino)->getArestas()->getCabeca();
253                          while (reversa != nullptr) {
254                               if (reversa->getDestino() == melhorVertice) {
255                                  arestaCoberta[reversa->getIdAresta()] = true;
256                                    arestasCobertas++;
257                                    break;
258                               }
259                               reversa = reversa->getProximo();
260                          }
261                     }
262                }
263                arestaAtual = arestaAtual->getProximo();
264           }
265      }
266
267      // Imprime a solucao do problema de cobertura de vertices
268     cout << endl << "*** Algoritmo Guloso para Cobertura de Vertices ***" << endl;
269     cout << "Quantidade de Vertices na solucao: " << qtdVerticesSolucao << endl;
270      cout << "Arestas cobertas: " << arestasCobertas << endl;
271      cout << "Numero de arestas no grafo: " << numArestasGrafo << endl;
272
273     /* Comentando a impressão do conjunto solução para evitar que o console trave
274      cout << "Conjunto solucao: { ";
275      for (int i = 1; i <= numVertices; i++) {
276           if (verticeEscolhido[i]) {
277               cout << i << " ";          // Imprime todos os vertices da solucao
278           }
279      }
280      cout << "}" << endl;
281      cout << "Quantidade de arestas cobertas: " << arestasCobertas << endl;
282      cout << "Quantidade de Arestas do grafo: " << numArestasGrafo << endl;
283      */
284
285      // Libera a memória alocada
286      delete[] verticeEscolhido;
287      delete[] arestaCoberta;
288      delete[] graus;
289      delete[] somaVizinhos;
290
291      // Calcula e imprime o tempo de execução
292      clock_t end = clock();
293      double duration = double(end - start) / CLOCKS_PER_SEC;
294      cout << "Tempo de execucao de alg_guloso_cobertura_vertices: " << duration
<< " segundos" << endl;
295
296      // Retorna a quantidade de vertices na solucao
297      return qtdVerticesSolucao;
298 }
```

int GrafoLista::alg_randomizado_cobertura_vertice () [override], [virtual]

Executa o algoritmo randomizado para cobertura de vértices.

Retorna

Resultado do algoritmo.

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

```
301                                                                 {
302       // Inicializa o tempo de execução
303       clock_t start = clock();
304 
305       unsigned seed = time(0);
306       srand(seed);
307 
308       int arestasCobertas = 0;
309      int qtdVerticesSolucao = 0; // Variável para contar a quantidade de vértices
na solução
310 
311       if (numVertices <= 0 || numArestasGrafo <= 0) {
312           cout << "Erro: O grafo deve conter vértices e arestas válidas." << endl;
313             return 0;
314       }
315 
316       bool* verticeEscolhido = new bool[numVertices + 1]();
317       bool* arestaCoberta = new bool[numArestasGrafo + 1]();
318       int* graus = new int[numVertices + 1]();
319       int* somaVizinhos = new int[numVertices + 1]();
320 
321       // Inicializa graus e soma dos vizinhos
322       for (int i = 1; i <= numVertices; i++) {
323            graus[i] = listaAdjVertices->getVertice(i)->getNumVizinhos();
324            NoAresta* arestaAtual =
listaAdjVertices->getVertice(i)->getArestas()->getCabeca();
325            while (arestaAtual != nullptr) {
326                 somaVizinhos[i] +=
listaAdjVertices->getVertice(arestaAtual->getDestino())->getNumVizinhos();
327                 arestaAtual = arestaAtual->getProximo();
328            }
329       }
330 
331       while (arestasCobertas < numArestasGrafo) {
332            int* candidatos = new int[numVertices + 1];
333            int numCandidatos = 0;
334            int melhorSoma = 0;
335            int melhorGrau = 0;
336 
337            // Seleciona vértices candidatos
338            for (int i = 1; i <= numVertices; i++) {
339                 if (!verticeEscolhido[i]) {
340                      int somaAtual = somaVizinhos[i];
341                      int grauAtual = graus[i];
342                     if ((somaAtual > melhorSoma) || (somaAtual == melhorSoma &&
grauAtual > melhorGrau)) {
343                           melhorSoma = somaAtual;
344                           melhorGrau = grauAtual;
345                           numCandidatos = 0;
346                           candidatos[numCandidatos++] = i;
347                      } else if (somaAtual == melhorSoma) {
348                           candidatos[numCandidatos++] = i;
349                      }
350                 }
351            }
```

```
352
353           if (numCandidatos == 0) {
354                delete[] candidatos;
355                break;
356           }
357
358           int janela = max(1, numCandidatos / 2); // Define o tamanho da janela
como metade dos candidatos
359           int escolhido = candidatos[rand() % janela];
360           delete[] candidatos;
361
362           verticeEscolhido[escolhido] = true;
363          qtdVerticesSolucao++; // Incrementa a quantidade de vértices na solução
364           NoAresta* arestaAtual =
listaAdjVertices->getVertice(escolhido)->getArestas()->getCabeca();
365           while (arestaAtual != nullptr) {
366                int destino = arestaAtual->getDestino();
367                int idAresta = arestaAtual->getIdAresta();
368                if (!arestaCoberta[idAresta]) {
369                     arestaCoberta[idAresta] = true;
370                     arestasCobertas++;
371                     graus[destino]--;
372                     somaVizinhos[destino] -= graus[escolhido];
373
374                     if (!direcionado) {
375                          // Marcar a aresta reversa como coberta
376                          NoAresta* reversa =
listaAdjVertices->getVertice(destino)->getArestas()->getCabeca();
377                          while (reversa != nullptr) {
378                               if (reversa->getDestino() == escolhido) {
379                                  arestaCoberta[reversa->getIdAresta()] = true;
380                                    arestasCobertas++;
381                                    break;
382                               }
383                               reversa = reversa->getProximo();
384                          }
385                     }
386                }
387                arestaAtual = arestaAtual->getProximo();
388           }
389      }
390
391      // Imprime a melhor solução encontrada
392     cout << endl << "*** Algoritmo Guloso Randomizado para Cobertura de Vertices
***" << endl;
393     cout << "Quantidade de Vertices na solucao: " << qtdVerticesSolucao << endl;
394      cout << "Arestas cobertas: " << arestasCobertas << endl;
395      cout << "Numero de arestas no grafo: " << numArestasGrafo << endl;
396
397      // Libera a memória alocada
398      delete[] verticeEscolhido;
399      delete[] arestaCoberta;
400      delete[] graus;
401      delete[] somaVizinhos;
402
403      // Calcula e imprime o tempo de execução
404      clock_t end = clock();
405      double duration = double(end - start) / CLOCKS_PER_SEC;
406    cout << "Tempo de execucao de alg_randomizado_cobertura_vertice: " << duration
<< " segundos" << endl;
407      return qtdVerticesSolucao;
408 }
```

int GrafoLista::alg_reativo_cobertura_vertice () `[override]`, `[virtual]`

Executa o algoritmo reativo para cobertura de vértices.

Retorna

Resultado do algoritmo.

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

```
410                                                                 {
411       // Inicializa o tempo de execução
412       clock_t start = clock();
413
414       unsigned seed = time(0);
415
416       srand(seed);
417       cout<<1+rand()%10<<endl;
418
419       const int numAlphas = 3;
420       float probabilidades[numAlphas] = {1.0 / numAlphas, 1.0 / numAlphas, 1.0 /
numAlphas};
421       float desempenho[numAlphas] = {0};
422       int escolhaAlpha = 0;
423       int arestasCobertas = 0;
424      int qtdVerticesSolucao = 0; // Variável para contar a quantidade de vértices
na solução
425
426       if (numVertices <= 0 || numArestasGrafo <= 0) {
427           cout << "Erro: O grafo deve conter vértices e arestas válidas." << endl;
428             return 0;
429       }
430
431       bool* verticeEscolhido = new bool[numVertices + 1]();
432       bool* arestaCoberta = new bool[numArestasGrafo + 1]();
433       int* graus = new int[numVertices + 1]();
434       int* somaVizinhos = new int[numVertices + 1]();
435
436       // Inicializa graus e soma dos vizinhos
437       for (int i = 1; i <= numVertices; i++) {
438            graus[i] = listaAdjVertices->getVertice(i)->getNumVizinhos();
439            NoAresta* arestaAtual =
listaAdjVertices->getVertice(i)->getArestas()->getCabeca();
440            while (arestaAtual != nullptr) {
441                 somaVizinhos[i] +=
listaAdjVertices->getVertice(arestaAtual->getDestino())->getNumVizinhos();
442                 arestaAtual = arestaAtual->getProximo();
443            }
444       }
445
446       int iteracao = 0;
447       while (arestasCobertas < numArestasGrafo) {
448            iteracao++;
449
450            // Ajusta as probabilidades dos alphas a cada 5 iterações
451            if (iteracao % 10 == 0) {
452                 float somaDesempenho = 0;
453                 for (int i = 0; i < numAlphas; i++) {
454                      somaDesempenho += desempenho[i] + 0.0001;
455                 }
456                 for (int i = 0; i < numAlphas; i++) {
457                   probabilidades[i] = (desempenho[i] + 0.0001) / somaDesempenho;
458                 }
459            }
460
461            // Escolhe um alpha com base nas probabilidades ajustadas
462            float r = (float)1+rand()%100 / RAND_MAX;
463            float acumulado = 0;
464            for (int i = 0; i < numAlphas; i++) {
465                 acumulado += probabilidades[i];
466                 if (r <= acumulado) {
```

```
467                     escolhaAlpha = i;
468                     break;
469                }
470           }
471
472           int* candidatos = new int[numVertices + 1];
473           int numCandidatos = 0;
474           int melhorSoma = 0;
475           int melhorGrau = 0;
476
477           // Seleciona vértices candidatos com base no alpha escolhido
478           for (int i = 1; i <= numVertices; i++) {
479                if (!verticeEscolhido[i]) {
480                     int somaAtual = somaVizinhos[i];
481                     int grauAtual = graus[i];
482                    if ((somaAtual > melhorSoma) || (somaAtual == melhorSoma &&
grauAtual > melhorGrau)) {
483                          melhorSoma = somaAtual;
484                          melhorGrau = grauAtual;
485                          numCandidatos = 0;
486                          candidatos[numCandidatos++] = i;
487                     } else if (somaAtual == melhorSoma) {
488                          candidatos[numCandidatos++] = i;
489                     }
490                }
491           }
492
493           if (numCandidatos == 0) {
494                delete[] candidatos;
495                break;
496           }
497
498           int janela = max(1, numCandidatos / 2); // Define o tamanho da janela
como metade dos candidatos
499           int escolhido = candidatos[rand() % janela];
500           delete[] candidatos;
501
502           verticeEscolhido[escolhido] = true;
503          qtdVerticesSolucao++; // Incrementa a quantidade de vértices na solução
504           NoAresta* arestaAtual =
listaAdjVertices->getVertice(escolhido)->getArestas()->getCabeca();
505           while (arestaAtual != nullptr) {
506                int destino = arestaAtual->getDestino();
507                int idAresta = arestaAtual->getIdAresta();
508                if (!arestaCoberta[idAresta]) {
509                     arestaCoberta[idAresta] = true;
510                     arestasCobertas++;
511                     graus[destino]--;
512                     somaVizinhos[destino] -= graus[escolhido];
513
514                     if (!direcionado) {
515                          // Marcar a aresta reversa como coberta
516                          NoAresta* reversa =
listaAdjVertices->getVertice(destino)->getArestas()->getCabeca();
517                          while (reversa != nullptr) {
518                               if (reversa->getDestino() == escolhido) {
519                                  arestaCoberta[reversa->getIdAresta()] = true;
520                                    arestasCobertas++;
521                                    break;
522                               }
523                               reversa = reversa->getProximo();
524                          }
525                     }
526                }
527                arestaAtual = arestaAtual->getProximo();
528           }
529
530           desempenho[escolhaAlpha] += 1.0 / numVertices;
531      }
```

```
532
533       // Imprime a melhor solução encontrada
534      cout << endl << "*** Algoritmo Guloso Reativo para Cobertura de Vertices ***"
<< endl;
535       cout << "Quantidade de Vertices na solucao reativa: " << qtdVerticesSolucao
<< endl;
536       cout << "Arestas cobertas: " << arestasCobertas << endl;
537       cout << "Numero de arestas no grafo: " << numArestasGrafo << endl;
538
539      /*  Comentando a impressão do conjunto solução para evitar que o console trave
540       cout << "Conjunto solucao: { ";
541       for (int i = 1; i <= numVertices; i++) {
542            if (verticeEscolhido[i]) {
543               cout << i << " ";            // Imprime todos os vertices da solucao
544            }
545       }
546       cout << "}" << endl;
547       cout << "Quantidade de Arestas cobertas: " << arestasCobertas << endl;
548       cout << "Quantidade de Arestas do grafo: " << numArestasGrafo << endl;
549       */
550
551       // Libera a memória alocada
552       delete[] verticeEscolhido;
553       delete[] arestaCoberta;
554       delete[] graus;
555       delete[] somaVizinhos;
556
557       // Calcula e imprime o tempo de execução
558       clock_t end = clock();
559       double duration = double(end - start) / CLOCKS_PER_SEC;
560       cout << "Tempo de execucao de alg guloso cobertura vertices: " << duration
<< " segundos" << endl;
561       return qtdVerticesSolucao;
562 }
```

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

Calcula a menor distância entre dois vértices.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

Retorna

Menor distância entre os vértices.

Implementa **Grafo** (*p.10*).

```
111                                                                              {
112       const int INF = 1000000;
113       int dist[numVertices + 1];
114       bool visitado[numVertices + 1];
115
116       for (int i = 1; i <= numVertices; i++) {
117            dist[i] = INF;
118            visitado[i] = false;
119       }
120
121       dist[origem] = 0;
122
123       for (int i = 1; i <= numVertices; i++) {
124            int u = -1;
125            for (int j = 1; j <= numVertices; j++) {
126                 if (!visitado[j] && (u == -1 || dist[j] < dist[u])) {
```

```
127                     u = j;
128                }
129           }
130
131           if (dist[u] == INF) {
132                break;
133           }
134
135           visitado[u] = true;
136           NoAresta* atual =
listaAdjVertices->getVertice(u)->getArestas()->getCabeca();
137           while (atual != nullptr) {
138                int v = atual->getDestino();
139                float peso = atual->getPeso();
140                if (dist[u] + peso < dist[v]) {
141                     dist[v] = dist[u] + peso;
142                }
143                atual = atual->getProximo();
144           }
145      }
146
147      return (dist[destino] == INF) ? -1 : dist[destino];
148 }
```

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

Executa a busca em profundidade (DFS) a partir de um vértice.

Parâmetros

id	Vértice inicial.
visitado	Vetor que marca os vértices visitados.

Reimplementa **Grafo** (*p.11*).

```
156                                                    {
157      if(listaAdjVertices->getVertice(id) == nullptr){
158           return;
159      }
160      visitado[id] = true;
161      NoAresta* atual =
this->listaAdjVertices->getVertice(id)->getArestas()->getCabeca();
162      while(atual != nullptr){
163           if(!visitado[atual->getDestino()]){
164                dfs(atual->getDestino(), visitado);
165           }
166           atual = atual->getProximo();
167      }
168 }
```

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

Verifica se um vértice existe no grafo.

Parâmetros

id	Identificador do vértice.

Retorna

True se o vértice existir, caso contrário, false.

Implementa **Grafo** (*p.12*).

```
25                                                  {
26       return this->listaAdjVertices->getVertice(id) != nullptr;
27 }
```

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

Retorna o número de vizinhos de um vértice.

Parâmetros

id	Identificador do vértice.

Retorna

Número de vértices vizinhos.

Reimplementa **Grafo** (*p.13*).

```
151                                                 {
152       return this->listaAdjVertices->getVertice(id)->getNumVizinhos();
153 }
```

void GrafoLista::imprimeGrafoLista ()

Imprime as informações do grafo.

```
179                                             {
180     cout << "                                                                    "
<< endl;
181       cout << endl << "--- Grafo Lista ---" << endl;
182     cout << "                                                                    "
<< endl << endl;
183       // Imprime as informacoes do grafo
184       imprimir_descricao();

185 }
```

void GrafoLista::imprimeListaAdj ()

Imprime a lista de adjacência do grafo.

```
173                                         {
174       // Imprime a lista de adjacencia
175       listaAdjVertices->imprimir();
176 }
```

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

Remove uma aresta do grafo.

Parâmetros

origem	Identificador do vértice de origem.

destino	Identificador do vértice de destino.

Reimplementa **Grafo** (*p.15*).

```
67                                                                          {
68      // Verifica se o vertice de origem existe
69      if(listaAdjVertices->getVertice(origem) == nullptr){
70           cout << "Erro: Vertice " << origem << " nao existe!" << endl;
                                /* { DEBUG } */
71           return;
72      }
73      // Verifica se o vertice de destino existe
74      if(listaAdjVertices->getVertice(destino) == nullptr){
75           cout << "Erro: Vertice " << destino <<" nao existe!" << endl;
                               /* { DEBUG } */
76           return;
77      }
78
79      // Remove a aresta
80      this->listaAdjVertices->remover_aresta(origem, destino);
81      if(!direcionado){
82           this->listaAdjVertices->remover_aresta(destino, origem);
83      }
84 }
```

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

Remove a primeira aresta associada a um vértice.

Parâmetros

id	Identificador do vértice.

Reimplementa **Grafo** (*p.15*).

```
87                                                                  {
88      // Verifica se o vertice existe
89      if(listaAdjVertices->getVertice(id) == nullptr){
90           cout << "Erro: Vertice " << id << " nao existe!" << endl;
                                /* { DEBUG } */
91           return;
92      }
93
94      // Remove a primeira aresta
95      this->listaAdjVertices->remover primeira aresta(id);
96 }
```

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

Remove um vértice do grafo.

Parâmetros

id	Identificador do vértice a ser removido.

Reimplementa **Grafo** (*p.16*).

```
99                                                  {
100      // Verifica se o vertice existe
101      if(!existe_vertice(id)){
```

```
102             cout << "Erro: Vertice " << id << " nao existe!" << endl;
                                    /* { DEBUG } */
103             return;
104         }
105
106         // Remove o vertice
107         this->listaAdjVertices->remover_vertice(id);
108 }
```

Atributos

ListaAdjVertice* GrafoLista::listaAdjVertices [protected]

Lista encadeada de vértices.

Referência da Classe GrafoMatriz

Representa um grafo utilizando uma matriz de adjacência.

`#include <GrafoMatriz.h>`

Diagrama de hierarquia da classe GrafoMatriz:

IMAGE

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 (número de conexões) de um vértice.

- int **get_num_vizinhos** (int id) override

Retorna o número de vizinhos de um vértice.

- void **dfs** (int id, bool *visitado) override

Executa a busca em profundidade (DFS) a partir de um vértice.

- bool **existe_vertice** (int id) override

Verifica se um vértice existe no grafo.

- void **adicionar_vertice** (int id, float peso=0.0) override

Adiciona um vértice ao grafo.

- void **adicionar_aresta** (int origem, int destino, float peso=1.0) override

Adiciona uma aresta ao grafo.

- void **remover_vertice** (int id) override

Remove um vértice 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 associada a um vértice.

- int **calcula_menor_dist** (int origem, int destino) override

Calcula a menor distância entre dois vértices.

- void **imprimirMatrizAdj** ()

Imprime a matriz de adjacência.

- void **imprimeGrafoMatriz** ()

Imprime os atributos do grafo.

- int **alg_guloso_cobertura_vertice** () override

Executa o algoritmo guloso para cobertura de vértices.

- int **alg_randomizado_cobertura_vertice** () override

Executa o algoritmo randomizado para cobertura de vértices.

- int **alg_reativo_cobertura_vertice** () override

Executa o algoritmo reativo para cobertura de vértices.

- int **alg_randomizado_cobertura_vertice_com_alpha** (double alpha)

Executa o algoritmo randomizado para cobertura de vértices utilizando um parâmetro alpha.

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 (número de vértices).

- bool **eh_direcionado** ()

Verifica se o grafo é direcionado.

- bool **vertice_ponderado** ()

Verifica se o grafo possui peso nos vértices.

- bool **aresta_ponderada** ()

Verifica se o grafo possui peso nas arestas.

- bool **eh_completo** ()

Verifica se o grafo é completo.

- void **imprimir_descricao** ()

Imprime os atributos básicos do grafo.

- void **imprimir_algoritmos_cobertura_vertice** (**Grafo** *grafo)

Imprime os resultados dos algoritmos gulosos para cobertura de vértices.

- void **analise_algoritmos_cobertura_vertice** (**Grafo** *grafo, int numVezes)

Analisa os algoritmos gulosos para cobertura de vértices.

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

Retorna os vértices vizinhos de um vértice.

- int **get_num_arestas_grafo** ()

Retorna o número de arestas do grafo.

- void **incrementa_num_vertices_grafo** ()

Incrementa o número de vértices do grafo.

- void **decrementa_num_vertices_grafo** ()

Decrementa o número de vértices do grafo.

- void **incrementa_num_arestas_grafos** ()

Incrementa o número de arestas do grafo.

- void **decrementa_num_arestas_grafos** ()

Decrementa o número de arestas do grafo.

- void **diminui_num_arestas_grafos** (int valor)

Diminui o número de arestas do grafo por um valor especificado.

- virtual int **calcula_maior_menor_dist** ()

Calcula a maior dentre as menores distâncias entre pares de vértices.

Atributos Protegidos

- int ** **matrizAdj**

Matriz de adjacência.

- int **tamanhoMatriz**

Tamanho da matriz (número de vértices, considerado MXM).

Atributos Protegidos herdados de Grafo

- int **numVertices**

Número de vértices (ordem) do grafo.

- int **numArestasGrafo**

Número de arestas do grafo.

- bool **direcionado**

Indica se o grafo é direcionado.

- bool **ponderadoVertices**

Indica se o grafo possui peso nos vértices.

- 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)

Carrega um grafo a partir de um arquivo.

Descrição detalhada

Representa um grafo utilizando uma matriz de adjacência.

A classe **GrafoMatriz** implementa as operações de manipulação e análise de grafos utilizando uma matriz para armazenar as conexões entre vértices.

Construtores e Destrutores

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

Construtor da classe **GrafoMatriz**.

Parâmetros

numVertices	Número de vértices do grafo.
direcionado	Indica se o grafo é direcionado.
ponderadoVertices	Indica se os vértices possuem peso.
ponderadoArestas	Indica se as arestas possuem peso.

```
13
                                                : Grafo(numVertices, direcionado,
ponderadoVertices, ponderadoArestas)
14 {
15      tamanhoMatriz = 10;
16
17      if (numVertices <= 0)
```

```
18        {
19             numVertices = 1;
20        }
21
22        while (tamanhoMatriz < numVertices)
23        {
24             tamanhoMatriz *= 2;
25        }
26
27        matrizAdj = new int *[tamanhoMatriz];
28        for (int i = 0; i < tamanhoMatriz; i++)
29        {
30             matrizAdj[i] = new int[tamanhoMatriz];
31             for (int j = 0; j < tamanhoMatriz; j++)
32             {
33                  matrizAdj[i][j] = 0; // Inicializa com 0 (sem aresta)
34             }
35        }
36 }
```

GrafoMatriz::~GrafoMatriz ()

Destrutor da classe **GrafoMatriz**.

```
39 {
40        if (matrizAdj != nullptr)
41        { // Verifica se ha memoria alocada
42             for (int i = 0; i < tamanhoMatriz; i++)
43             {
44                  delete[] matrizAdj[i];
45             }
46             delete[] matrizAdj;
47             matrizAdj = nullptr; // Evita ponteiro danificado
48        }
49 }
```

Documentação das funções

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

Adiciona uma aresta ao grafo.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.
peso	Peso da aresta (valor padrão 1.0).

Reimplementa **Grafo** (*p.6*).

```
194 {
195      if (origem < 0 || origem > numVertices || destino < 0 || destino > numVertices)
196        {
197             cout << "Erro: indices de origem ou destino invalidos.\n";
198             return;
199        }
200        if( origem == destino){
201             cout << "Erro: origem e destino iguais.\n";
```

```
202             return;
203         }
204
205         if(matrizAdj[origem][destino] != 0){
206             cout << "Erro: aresta ja existe.\n";
207             return;
208         }
209
210
211         if (ponderadoArestas)
212         {
213             matrizAdj[origem][destino] = peso;
214             incrementa num arestas grafos();
215             if (!direcionado)
216             {
217                 matrizAdj[destino][origem] = peso;
218                 incrementa_num_arestas_grafos();
219             }
220         }
221         else
222         {
223             matrizAdj[origem][destino] = 1;
224             incrementa num arestas grafos();
225             if (!direcionado)
226             {
227                 matrizAdj[destino][origem] = 1;
228                 incrementa_num_arestas_grafos();
229             }
230         }
231 }
```

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

Adiciona um vértice ao grafo.

Parâmetros

id	Identificador do novo vértice.
peso	Peso do vértice (valor padrão 0.0).

Reimplementa **Grafo** (*p.7*).

```
55 {
56       int novoNumVertices = numVertices + 1;
57       numVertices = novoNumVertices;
58
59       if (novoNumVertices >= tamanhoMatriz)
60       {
61            cout << "Aumentando matriz de adjacencia...\n";
62            int antigoTamanhoMatriz = tamanhoMatriz;
63          tamanhoMatriz = max(tamanhoMatriz * 2, novoNumVertices); // Expansao mais
segura
64            int **novaMatriz = new int *[tamanhoMatriz];
65
66            for (int i = 0; i < tamanhoMatriz; i++)
67            {
68                 novaMatriz[i] = new int[tamanhoMatriz](); // Inicializa com 0
69            }
70
71            // Copia os valores da matriz antiga para a nova matriz
72            for (int i = 0; i < antigoTamanhoMatriz; i++)
73            {
74                 for (int j = 0; j < antigoTamanhoMatriz; j++)
75                 {
76                      novaMatriz[i][j] = matrizAdj[i][j];
```

```
77                }
78           }
79
80           // Libera a matriz antiga corretamente
81           for (int i = 0; i < antigoTamanhoMatriz; i++) // Correcao aqui
82           {
83                delete[] matrizAdj[i];
84           }
85           delete[] matrizAdj;
86
87           matrizAdj = novaMatriz;
88      }
89 }
```

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

Executa o algoritmo guloso para cobertura de vértices.

Retorna

Resultado do algoritmo (por exemplo, tamanho da cobertura).

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

```
380                                                                {
381      // Inicializa o tempo de execução
382      clock_t start = clock();
383
384      // Aloca estruturas básicas
385      bool *verticeEscolhido = new bool[numVertices + 1];
386      bool **arestaCoberta = new bool*[numVertices + 1];
387      int *graus = new int[numVertices + 1];
388      int qtdVerticesSolucao = 0;
389
390      // Aloca estruturas para listas de vizinhos
391      int **vizinhos = new int*[numVertices + 1];
392      int *contVizinhos = new int[numVertices + 1];   // Quantos vizinhos foram
armazenados para cada vértice
393
394      // Inicializa estruturas
395      for (int i = 1; i <= numVertices; i++) {
396           verticeEscolhido[i] = false;
397           arestaCoberta[i] = new bool[numVertices + 1];
398           for (int j = 1; j <= numVertices; j++) {
399                arestaCoberta[i][j] = false;
400           }
401           graus[i] = get_num_vizinhos(i);
402           contVizinhos[i] = 0;
403         // Aloca espaço para a lista de vizinhos com tamanho igual ao grau inicial
404           if (graus[i] > 0)
405                vizinhos[i] = new int[graus[i]];
406           else
407                vizinhos[i] = nullptr;
408      }
409
410      // Preenche as listas de vizinhos com base na matriz de adjacência
411      for (int i = 1; i <= numVertices; i++) {
412           for (int j = 1; j <= numVertices; j++) {
413                if (matrizAdj[i][j] != 0) {
414                     vizinhos[i][contVizinhos[i]++] = j;
415                }
416           }
417      }
418
```

```
419       // Calcula o total de "entradas de arestas" (para grafos não direcionados,
cada aresta aparece duas vezes)
420       int totalEdges = numArestasGrafo;
421
422       int arestasCobertas = 0;
423
424       // Função lambda que calcula a soma dos graus dos vizinhos não cobertos de
v
425       auto calcularSomaVizinhos = [&](int v) {
426            int soma = 0;
427            for (int k = 0; k < contVizinhos[v]; k++) {
428                 int u = vizinhos[v][k];
429                 if (!verticeEscolhido[u] && !arestaCoberta[v][u])
430                      soma += graus[u];
431            }
432            return soma;
433       };
434
435       // Loop principal do algoritmo guloso
436       while (true) {
437            int melhorVertice = -1;
438            int melhorSoma = -1;
439            int melhorGrau = -1;
440
441            for (int i = 1; i <= numVertices; i++) {
442                 if (!verticeEscolhido[i]) {
443                      int somaAtual = calcularSomaVizinhos(i);
444                      int grauAtual = graus[i];
445                    // Critério de desempate: maior soma, depois maior grau e, por
fim, menor índice
446                      if (somaAtual > melhorSoma ||
447                          (somaAtual == melhorSoma && grauAtual > melhorGrau) ||
448                          (somaAtual == melhorSoma && grauAtual == melhorGrau &&
(melhorVertice == -1 || i < melhorVertice))) {
449                           melhorSoma = somaAtual;
450                           melhorGrau = grauAtual;
451                           melhorVertice = i;
452                      }
453                 }
454            }
455
456            if (melhorVertice == -1)
457                 break;
458
459            verticeEscolhido[melhorVertice] = true;
460            qtdVerticesSolucao++;
461          // Atualiza cobertura das arestas a partir do vértice escolhido, iterando
somente sobre seus vizinhos
462            for (int k = 0; k < contVizinhos[melhorVertice]; k++) {
463                 int j = vizinhos[melhorVertice][k];
464                 if (!arestaCoberta[melhorVertice][j]) {
465                      arestaCoberta[melhorVertice][j] = true;
466                      arestasCobertas++;
467                      if (!direcionado) {
468                           if (!arestaCoberta[j][melhorVertice]) {
469                                arestaCoberta[j][melhorVertice] = true;
470                                arestasCobertas++;
471                           }
472                           // Atualiza o grau do vizinho
473                           graus[j]--;
474                      }
475                 }
476            }
477
478            // Se todas as arestas estiverem cobertas, encerra o loop
479            if (arestasCobertas == totalEdges)
480                 break;
481       }
482
```

```
483       // Imprime o conjunto solução
484      cout << endl << "*** Algoritmo Guloso para Cobertura de Vertices ***" << endl;
485      cout << "Quantidade de vertices na solucao: " << qtdVerticesSolucao << endl;
486
487      /* Comentando a impressão do conjunto solução para evitar que o console trave
488       cout << "Conjunto solucao: { ";
489       for (int i = 1; i <= numVertices; i++) {
490            if (verticeEscolhido[i]) {
491                cout << i << " ";          // Imprime todos os vertices da solucao
492            }
493       }
494       cout << "}" << endl;
495       cout << "Quantidade de arestas cobertas: " << arestasCobertas << endl;
496       cout << "Quantidade de Arestas do grafo: " << numArestasGrafo << endl;
497       */
498
499       // Libera a memória alocada
500       for (int i = 1; i <= numVertices; i++) {
501            delete[] arestaCoberta[i];
502            if (vizinhos[i] != nullptr)
503                 delete[] vizinhos[i];
504       }
505       delete[] arestaCoberta;
506       delete[] verticeEscolhido;
507       delete[] graus;
508       delete[] vizinhos;
509       delete[] contVizinhos;
510
511       // Calcula e imprime o tempo de execução
512       clock_t end = clock();
513       double duration = double(end - start) / CLOCKS_PER_SEC;
514      cout << "Tempo de execucao de alg guloso cobertura vertice: " << duration <<
" segundos" << endl;
515       return qtdVerticesSolucao;
516 }
```

int GrafoMatriz::alg_randomizado_cobertura_vertice () `[override]`, `[virtual]`

Executa o algoritmo randomizado para cobertura de vértices.

Retorna

Resultado do algoritmo.

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

```
519                                                                          {
520       clock_t start = clock();
521
522      double alpha = 0.3 + (rand() % 40) / 100.0; // Define um alpha aleatório entre
0.3 e 0.7
523      srand(time(nullptr)); // Garante que a escolha aleatória seja diferente a cada
execução
524
525       bool *verticeEscolhido = new bool[numVertices + 1];
526       bool **arestaCoberta = new bool*[numVertices + 1];
527       int *graus = new int[numVertices + 1];
528       int qtdVerticesSolucao = 0;
529       int **vizinhos = new int*[numVertices + 1];
530       int *contVizinhos = new int[numVertices + 1];
531
532       for (int i = 1; i <= numVertices; i++) {
533            verticeEscolhido[i] = false;
534            arestaCoberta[i] = new bool[numVertices + 1];
535            for (int j = 1; j <= numVertices; j++) {
```

```
536                 arestaCoberta[i][j] = false;
537           }
538           graus[i] = get_num_vizinhos(i);
539           contVizinhos[i] = 0;
540           vizinhos[i] = (graus[i] > 0) ? new int[graus[i]] : nullptr;
541      }
542
543      for (int i = 1; i <= numVertices; i++) {
544           for (int j = 1; j <= numVertices; j++) {
545                if (matrizAdj[i][j] != 0) {
546                     vizinhos[i][contVizinhos[i]++] = j;
547                }
548           }
549      }
550
551      int totalEdges = numArestasGrafo;
552      int arestasCobertas = 0;
553
554      auto calcularSomaVizinhos = [&](int v) {
555           int soma = 0;
556           for (int k = 0; k < contVizinhos[v]; k++) {
557                int u = vizinhos[v][k];
558                if (!verticeEscolhido[u] && !arestaCoberta[v][u])
559                     soma += graus[u];
560           }
561           return soma;
562      };
563
564      while (arestasCobertas < totalEdges) {
565           int *candidatos = new int[numVertices];
566           int *somas = new int[numVertices];
567           int numCandidatos = 0;
568           int melhorSoma = -1;
569
570           for (int i = 1; i <= numVertices; i++) {
571                if (!verticeEscolhido[i]) {
572                     int somaAtual = calcularSomaVizinhos(i);
573                     if (somaAtual > melhorSoma)
574                          melhorSoma = somaAtual;
575                     candidatos[numCandidatos] = i;
576                     somas[numCandidatos] = somaAtual;
577                     numCandidatos++;
578                }
579           }
580
581           int *listaRestrita = new int[numCandidatos];
582           int numRestritos = 0;
583           for (int i = 0; i < numCandidatos; i++) {
584                if (somas[i] >= melhorSoma * alpha) {
585                     listaRestrita[numRestritos++] = candidatos[i];
586                }
587           }
588
589           delete[] candidatos;
590           delete[] somas;
591
592           if (numRestritos == 0) {
593                delete[] listaRestrita;
594                break;
595           }
596
597           int escolhido = listaRestrita[rand() % numRestritos];
598           delete[] listaRestrita;
599
600           verticeEscolhido[escolhido] = true;
601           qtdVerticesSolucao++;
602
603           for (int k = 0; k < contVizinhos[escolhido]; k++) {
604                int j = vizinhos[escolhido][k];
```

```
605                 if (!arestaCoberta[escolhido][j]) {
606                     arestaCoberta[escolhido][j] = true;
607                     arestasCobertas++;
608                     if (!direcionado) {
609                         if (!arestaCoberta[j][escolhido]) {
610                             arestaCoberta[j][escolhido] = true;
611                             arestasCobertas++;
612                         }
613                         graus[j]--;
614                     }
615                 }
616             }
617         }
618 
619         cout << endl << "*** Algoritmo Guloso Randomizado para Cobertura de Vertices
***" << endl;
620         cout << "Quantidade de vertices na solucao: " << qtdVerticesSolucao << endl;
621 
622         for (int i = 1; i <= numVertices; i++) {
623             delete[] arestaCoberta[i];
624             if (vizinhos[i] != nullptr)
625                 delete[] vizinhos[i];
626         }
627         delete[] arestaCoberta;
628         delete[] verticeEscolhido;
629         delete[] graus;
630         delete[] vizinhos;
631         delete[] contVizinhos;
632 
633         clock_t end = clock();
634         double duration = double(end - start) / CLOCKS_PER_SEC;
635         cout << "Tempo de execucao de alg_randomizado_cobertura_vertice: " << duration
<< " segundos" << endl;
636         return qtdVerticesSolucao;
637 }
```

int GrafoMatriz::alg_randomizado_cobertura_vertice_com_alpha (double alpha)

Executa o algoritmo randomizado para cobertura de vértices utilizando um parâmetro alpha.

Parâmetros

alpha	Parâmetro que influencia a aleatoriedade do algoritmo.

Retorna

Resultado do algoritmo.

```
738 
            {
739         // Aloca e inicializa estruturas
740         bool *verticeEscolhido = new bool[numVertices + 1];
741         bool **arestaCoberta = new bool*[numVertices + 1];
742         int *graus = new int[numVertices + 1];
743         int qtdVerticesSolucao = 0;
744         int **vizinhos = new int*[numVertices + 1];
745         int *contVizinhos = new int[numVertices + 1];
746 
747         // Inicializa as estruturas
748         for (int i = 1; i <= numVertices; i++) {
749             verticeEscolhido[i] = false;
750             arestaCoberta[i] = new bool[numVertices + 1];
751             for (int j = 1; j <= numVertices; j++) {
752                 arestaCoberta[i][j] = false;
```

```
753             }
754             graus[i] = get num vizinhos(i); // Obtém o grau do vértice i
755             contVizinhos[i] = 0;
756             if (graus[i] > 0) {
757                 vizinhos[i] = new int[graus[i]]; // Aloca espaço para os vizinhos
758             } else {
759                 vizinhos[i] = nullptr;
760             }
761         }
762
763         // Preenche as listas de vizinhos com base na matriz de adjacência
764         for (int i = 1; i <= numVertices; i++) {
765             for (int j = 1; j <= numVertices; j++) {
766                 if (matrizAdj[i][j] != 0) {
767                     vizinhos[i][contVizinhos[i]++] = j;
768                 }
769             }
770         }
771
772         // Calcula o total de arestas no grafo
773         int totalEdges = numArestasGrafo;
774         int arestasCobertas = 0;
775
776         // Função lambda para calcular a soma dos graus dos vizinhos não cobertos
777         auto calcularSomaVizinhos = [&](int v) {
778             int soma = 0;
779             for (int k = 0; k < contVizinhos[v]; k++) {
780                 int u = vizinhos[v][k];
781                 if (!verticeEscolhido[u] && !arestaCoberta[v][u]) {
782                     soma += graus[u];
783                 }
784             }
785             return soma;
786         };
787
788         // Loop principal do algoritmo guloso randomizado
789         while (arestasCobertas < totalEdges) {
790             // Cria lista de candidatos e suas somas de vizinhos
791             int *candidatos = new int[numVertices];
792             int *somas = new int[numVertices];
793             int numCandidatos = 0;
794             int melhorSoma = -1;
795
796             // Encontra todos os vértices não escolhidos e calcula suas somas
797             for (int i = 1; i <= numVertices; i++) {
798                 if (!verticeEscolhido[i]) {
799                     int somaAtual = calcularSomaVizinhos(i);
800                     if (somaAtual > melhorSoma) {
801                         melhorSoma = somaAtual;
802                     }
803                     candidatos[numCandidatos] = i;
804                     somas[numCandidatos] = somaAtual;
805                     numCandidatos++;
806                 }
807             }
808
809             // Cria a lista restrita de candidatos com base no alpha
810             int *listaRestrita = new int[numCandidatos];
811             int numRestritos = 0;
812             for (int i = 0; i < numCandidatos; i++) {
813                 if (somas[i] >= melhorSoma * alpha) {
814                     listaRestrita[numRestritos++] = candidatos[i];
815                 }
816             }
817
818             // Libera a memória dos arrays temporários
819             delete[] candidatos;
820             delete[] somas;
821
```

```
822             // Se não houver candidatos, encerra o loop
823             if (numRestritos == 0) {
824                  delete[] listaRestrita;
825                  break;
826             }
827
828             // Escolhe um vértice aleatório da lista restrita
829             int escolhido = listaRestrita[rand() % numRestritos];
830             delete[] listaRestrita;
831
832             // Marca o vértice escolhido como parte da solução
833             verticeEscolhido[escolhido] = true;
834             qtdVerticesSolucao++;
835
836             // Cobre as arestas incidentes ao vértice escolhido
837             for (int k = 0; k < contVizinhos[escolhido]; k++) {
838                  int j = vizinhos[escolhido][k];
839                  if (!arestaCoberta[escolhido][j]) {
840                       arestaCoberta[escolhido][j] = true;
841                       arestasCobertas++;
842                       if (!direcionado) {
843                            if (!arestaCoberta[j][escolhido]) {
844                                 arestaCoberta[j][escolhido] = true;
845                                 arestasCobertas++;
846                            }
847                            // Atualiza o grau do vizinho
848                            graus[j]--;
849                       }
850                  }
851             }
852        }
853
854        // Libera a memória alocada
855        for (int i = 1; i <= numVertices; i++) {
856             delete[] arestaCoberta[i];
857             if (vizinhos[i] != nullptr) {
858                  delete[] vizinhos[i];
859             }
860        }
861        delete[] arestaCoberta;
862        delete[] verticeEscolhido;
863        delete[] graus;
864        delete[] vizinhos;
865        delete[] contVizinhos;
866
867        // Retorna o número de vértices na solução
868        return qtdVerticesSolucao;
869   }
```

int GrafoMatriz::alg_reativo_cobertura_vertice () `[override]`, `[virtual]`

Executa o algoritmo reativo para cobertura de vértices.

Retorna

Resultado do algoritmo.

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

```
640                                                                    {
641        clock_t start = clock();
642
643        // Parâmetros internos
644        const int numAlphas = 5; // Reduzido de 10 para 5
645        const int maxIteracoes = 10; // Mantido em 10 iterações
```

```
646       const int blocoIteracoes = 5; // Ajuste a cada 5 iterações
647
648       // Valores de alpha (variando de 0.1 a 0.9)
649       double alphas[numAlphas];
650       for (int i = 0; i < numAlphas; i++) {
651            alphas[i] = 0.1 + (0.8 * i) / (numAlphas - 1);
652       }
653
654       // Probabilidades iniciais (uniforme)
655       double probabilidades[numAlphas];
656       for (int i = 0; i < numAlphas; i++) {
657            probabilidades[i] = 1.0 / numAlphas;
658       }
659
660       // Estruturas para armazenar o desempenho de cada alpha
661       double somaQualidade[numAlphas] = {0};
662       int contagemAlpha[numAlphas] = {0};
663
664       // Melhor solução encontrada
665       int qtdVerticesSolucao = numVertices; // Substitui INT_MAX
666       int melhorIteracao = -1;
667
668       // Loop principal do algoritmo reativo
669       for (int iteracao = 1; iteracao <= maxIteracoes; iteracao++) {
670            // Escolhe um alpha com base nas probabilidades atuais
671            double r = (double)rand() / RAND MAX;
672            double somaProbabilidades = 0;
673            int alphaEscolhido = 0;
674            for (int i = 0; i < numAlphas; i++) {
675                 somaProbabilidades += probabilidades[i];
676                 if (r <= somaProbabilidades) {
677                      alphaEscolhido = i;
678                      break;
679                 }
680            }
681
682            // Executa o algoritmo guloso randomizado com o alpha escolhido
683            int qtdVerticesAtual =
alg randomizado cobertura vertice com alpha(alphas[alphaEscolhido]);
684
685            // Atualiza o desempenho do alpha escolhido
686            somaQualidade[alphaEscolhido] += qtdVerticesAtual;
687            contagemAlpha[alphaEscolhido]++;
688
689            // Atualiza a melhor solução encontrada
690            if (qtdVerticesAtual < qtdVerticesSolucao) {
691                 qtdVerticesSolucao = qtdVerticesAtual;
692                 melhorIteracao = iteracao;
693            }
694
695            // Ajusta as probabilidades após cada bloco de iterações
696            if (iteracao % blocoIteracoes == 0) {
697                 double mediaQualidade[numAlphas];
698                 for (int i = 0; i < numAlphas; i++) {
699                      if (contagemAlpha[i] > 0) {
700                        mediaQualidade[i] = somaQualidade[i] / contagemAlpha[i];
701                      } else {
702                         mediaQualidade[i] = numVertices; // Penaliza alphas não
utilizados
703                      }
704                 }
705
706                 // Calcula a qualidade relativa de cada alpha
707                 double somaInversos = 0;
708                 for (int i = 0; i < numAlphas; i++) {
709                      somaInversos += 1.0 / mediaQualidade[i];
710                 }
711
712                 // Atualiza as probabilidades
```

```
713                 for (int i = 0; i < numAlphas; i++) {
714                    probabilidades[i] = (1.0 / mediaQualidade[i]) / somaInversos;
715                 }
716 
717                 // Reinicia as estruturas de desempenho
718                 for (int i = 0; i < numAlphas; i++) {
719                      somaQualidade[i] = 0;
720                      contagemAlpha[i] = 0;
721                 }
722            }
723       }
724 
725       // Imprime a melhor solução encontrada
726      cout << endl << "*** Algoritmo Guloso Reativo para Cobertura de Vertices ***"
<< endl;
727      cout << "Quantidade de vertices na solucao: " << qtdVerticesSolucao << endl;
728      cout << "Melhor solucao encontrada na iteracao: " << melhorIteracao << endl;
729 
730       // Calcula e imprime o tempo de execução
731       clock_t end = clock();
732       double duration = double(end - start) / CLOCKS_PER_SEC;
733      cout << "Tempo de execucao de alg_reativo_cobertura_vertice: " << duration
<< " segundos" << endl;
734 
735       return qtdVerticesSolucao;
736 }
```

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

Calcula a menor distância entre dois vértices.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

Retorna

Menor distância entre os vértices.

Implementa **Grafo** (*p.10*).

```
235 {
236       const int INF = 1000000; // Valor grande para representar infinito
237       int *dist = new int[numVertices + 1];
238       int *prev = new int[numVertices + 1];
239       bool *visitado = new bool[numVertices + 1];
240 
241       for (int i = 1; i <= numVertices; i++)
242       {
243            dist[i] = INF;
244            prev[i] = -1;
245            visitado[i] = false;
246       }
247 
248       dist[origem] = 0;
249 
250       // Loop principal do algoritmo de Dijkstra
251       for (int i = 1; i <= numVertices; i++)
252       {
253            int u = -1;
254 
255            // Encontra o vertice nao visitado com a menor distancia
256            for (int j = 1; j <= numVertices; j++)
257            {
258                 if (!visitado[j] && (u == -1 || dist[j] < dist[u]))
```

```
259                 {
260                     u = j;
261                 }
262             }
263 
264             // Se a menor distancia e infinita, todos os vertices restantes sao
inacessiveis
265             if (dist[u] == INF)
266             {
267                 break;
268             }
269 
270             visitado[u] = true;
271 
272             // Atualiza as distancias dos vizinhos do vertice atual
273             for (int v = 1; v <= numVertices; v++)
274             {
275                 if (matrizAdj[u][v] != 0 && dist[u] + matrizAdj[u][v] < dist[v])
276                 {
277                     dist[v] = dist[u] + matrizAdj[u][v];
278                     prev[v] = u;
279                 }
280             }
281         }
282 
283         int menorDist = dist[destino];
284 
285         delete[] dist;
286         delete[] prev;
287         delete[] visitado;
288 
289         return menorDist;
290 }
```

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

Executa a busca em profundidade (DFS) a partir de um vértice.

Parâmetros

id	Vértice inicial.
visitado	Vetor que marca os vértices visitados.

Reimplementa **Grafo** (*p.11*).

```
313 {
314         if (id < 0 || id > numVertices || visitado[id])
315         {
316             return; // Verifica se o vertice eh valido ou ja foi visitado
317         }
318 
319         visitado[id] = true; // Marca o vertice como visitado
320 
321         for (int i = 0; i <= numVertices; i++)
322         {
323             if (matrizAdj[id][i] != 0 && !visitado[i])
324             { // Se houver uma aresta e o vertice nao foi visitado
325                 dfs(i, visitado);
326             }
327         }
328 }
```

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

Verifica se um vértice existe no grafo.

Parâmetros

id	Identificador do vértice.

Retorna

True se o vértice existir, caso contrário, false.

Implementa **Grafo** (*p.12*).

```
332 {
333       return (id >= 0 && id <= numVertices);
334 }
```

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

Retorna o número de vizinhos de um vértice.

Parâmetros

id	Identificador do vértice.

Retorna

Número de vizinhos.

Reimplementa **Grafo** (*p.13*).

```
294 {
295       if (id < 0 || id > numVertices)
296       {
297            return 0; // Retorna 0 se o ID do vertice for invalido
298       }
299
300       int contador = 0;
301       for (int i = 0; i <= numVertices; i++)
302       {
303            if (matrizAdj[id][i] != 0)
304            { // Se houver uma aresta entre os vertices
305                 contador++;
306            }
307       }
308       return contador;
309 }
```

int GrafoMatriz::grauVertice (int vertice)

Retorna o grau (número de conexões) de um vértice.

Parâmetros

vertice	Identificador do vértice.

Retorna

Grau do vértice.

void GrafoMatriz::imprimeGrafoMatriz ()

Imprime os atributos do grafo.

```
368 {
369     cout << "                                                                    "
<< endl;
370       cout << endl
371             << "--- Grafo Matriz ---" << endl;
372     cout << "____________________________________________________________________"
<< endl
373             << endl;
374       imprimir descricao();
375 }
```

void GrafoMatriz::imprimirMatrizAdj ()

Imprime a matriz de adjacência.

```
340 {
341     cout << "                                                                    "
<< endl;
342       cout << endl << "--- Matriz de Adjacencia ---" << endl << endl;
343
344       // Imprime o cabecalho dos indices dos vertices
345       cout << "   V";
346       for (int i = 0; i < tamanhoMatriz; i++)
347       {
348            cout << std::setw(3) << i << " ";
349       }
350       cout << endl;
351
352       for (int i = 0; i < tamanhoMatriz; i++)
353       {
354            // Imprime o indice do vertice na lateral esquerda
355            cout << std::setw(3) << i;
356
357            for (int j = 0; j < tamanhoMatriz; j++)
358            {
359                 std::cout << std::setw(3) << matrizAdj[i][j] << " ";
360            }
361            std::cout << std::endl;
362       }
363     cout << "                                                                    "
<< endl;
364 }
```

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

Remove uma aresta do grafo.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

Reimplementa **Grafo** (*p.15*).

```
152 {
153     if (origem < 0 || origem >= numVertices || destino < 0 || destino >= numVertices)
154       {
155            cout << "Erro: indices de origem ou destino invalidos.\n";
156            return;
157       }
158
159       matrizAdj[origem][destino] = 0;
160       decrementa_num_arestas_grafos();
161       if (!direcionado)
162       {
163            matrizAdj[destino][origem] = 0;
164            decrementa_num_arestas_grafos();
165       }
166 }
```

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

Remove a primeira aresta associada a um vértice.

Parâmetros

id	Identificador do vértice.

Reimplementa **Grafo** (*p.15*).

```
170 {
171       if (id < 0 || id > numVertices)
172       {
173            cout << "Erro: indice de vertice invalido." << endl;
174            return;
175       }
176
177       for (int i = 0; i <= numVertices; i++)
178       {
179            if (matrizAdj[id][i] != 0)
180            {
181                 matrizAdj[id][i] = 0;
182                 decrementa num arestas grafos();
183                 if (!direcionado)
184                 {
185                      matrizAdj[i][id] = 0;
186                      decrementa_num_arestas_grafos();
187                 }
188                 return;
189            }
190       }
191 }
```

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

Remove um vértice do grafo.

Parâmetros

id	Identificador do vértice a ser removido.

Reimplementa **Grafo** (*p.16*).

```
93 {
94       if (id < 0 || id >= numVertices)
```

```
95      {
96           cout << "Erro: indice de vertice invalido.\n";
97           return;
98      }
99      // Atualiza o contador de arestas
100      for (int i = 0; i < numVertices; i++)
101      {
102           if (matrizAdj[id][i] != 0)
103           {
104                decrementa_num_arestas_grafos();
105                if (!direcionado)
106                {
107                     decrementa_num_arestas_grafos();
108                }
109           }
110           if (matrizAdj[i][id] != 0 && i != id)
111           {
112                decrementa num arestas grafos();
113           }
114      }
115      int novoNumVertices = numVertices - 1;
116      int **novaMatriz = new int *[tamanhoMatriz];
117
118      for (int i = 0; i < tamanhoMatriz; i++)
119      {
120           novaMatriz[i] = new int[tamanhoMatriz]();
121      }
122
123     // Copia os valores da matriz antiga para a nova, excluindo o vertice removido
124      for (int i = 0, ni = 0; i < tamanhoMatriz; i++)
125      {
126           if (i == id)
127                continue;
128
129           for (int j = 0, nj = 0; j < tamanhoMatriz; j++)
130           {
131                if (j == id)
132                     continue;
133
134                novaMatriz[ni][nj] = matrizAdj[i][j];
135                nj++;
136           }
137           ni++;
138      }
139
140      // Libera a matriz antiga
141      for (int i = 0; i < numVertices; i++)
142      {
143           delete[] matrizAdj[i];
144      }
145      delete[] matrizAdj;
146
147      matrizAdj = novaMatriz;
148      numVertices = novoNumVertices;
149 }
```

Atributos

int** GrafoMatriz::matrizAdj [protected]

Matriz de adjacência.

int GrafoMatriz::tamanhoMatriz `[protected]`

Tamanho da matriz (número de vértices, considerado MXM).

Referência da Classe ListaAdjAresta

Gerencia a lista encadeada de arestas associadas a um vértice.

`#include <ListaAdjAresta.h>`

Membros Públicos

- **ListaAdjAresta** (**GrafoLista** *grafo)

*Construtor da classe **ListaAdjAresta**.*

- **~ListaAdjAresta** ()

*Destrutor da classe **ListaAdjAresta**.*

- **NoAresta** * **getCabeca** ()

Retorna o primeiro nó (cabeça) da lista de arestas.

- int **getNumVerticesVizinhos** ()

Retorna a quantidade de vértices vizinhos (número de arestas na lista).

- void **adicionar_aresta** (int origem, int destino, float peso)

Adiciona uma aresta à lista.

- void **remover_aresta** (int origem, int destino)

Remove uma aresta específica da lista.

- void **remover_primeira_aresta** ()

Remove a primeira aresta da lista.

- int **getIdAresta** (int destino)

Retorna o identificador da aresta que termina no vértice destino.

Descrição detalhada

Gerencia a lista encadeada de arestas associadas a um vértice.

Construtores e Destrutores

ListaAdjAresta::ListaAdjAresta (GrafoLista * grafo)

Construtor da classe **ListaAdjAresta**.

Parâmetros

grafo	Ponteiro para o objeto **GrafoLista** que utiliza essa lista.

```
9                                                                   {
10      this->grafo = grafo;
11      this->cabeca = nullptr;
12 }
```

ListaAdjAresta::~ListaAdjAresta ()

Destrutor da classe **ListaAdjAresta**.

```
14                                              {
15      // Desaloca a memoria para cada no da lista de arestas
16      NoAresta* atual = this->cabeca;
17      while (atual != nullptr) {
18           NoAresta* next = atual->getProximo();
19           delete atual;
20           atual = next;
21      }
22 }
```

Documentação das funções

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

Adiciona uma aresta à lista.

Parâmetros

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

```
32
            {
33      // Verifica se a aresta ja existe
34      NoAresta* atual = this->cabeca;
35      while (atual != nullptr) {
36           if (atual->getOrigem() == origem && atual->getDestino() == destino) {
37             //cout << "Erro: Aresta " << origem << " -> " << destino << " ja existe.
Nao eh possivel multiarestas" <<   endl;                              /* { DEBUG } */
38                return;
39           }
40           atual = atual->getProximo();
41      }
42
43      // Adiciona uma nova aresta
44      grafo->incrementa_num_arestas_grafos();
45      NoAresta* novaAresta = new NoAresta(origem, destino, peso,
grafo->get_num_arestas_grafo());
46
47      novaAresta->setProximo(this->cabeca);
48      this->cabeca = novaAresta;
49      //cout << "Adicionada Aresta(" << novaAresta->getIdAresta() << "): " <<
novaAresta->getOrigem() << " -> " << novaAresta->getDestino() << endl;           /*
{ DEBUG } */
50 }
```

NoAresta * ListaAdjAresta::getCabeca ()

Retorna o primeiro nó (cabeça) da lista de arestas.

Retorna

Ponteiro para o nó inicial da lista.

```
27                                                  {
28       return this->cabeca;
29  }
```

int ListaAdjAresta::getIdAresta (int destino)

Retorna o identificador da aresta que termina no vértice destino.

Parâmetros

destino	Identificador do vértice de destino.

Retorna

ID da aresta.

```
105                                                      {
106       NoAresta* atual = this->cabeca;
107       while (atual != nullptr) {
108            if (atual->getDestino() == destino) {
109                 return atual->getIdAresta();
110            }
111            atual = atual->getProximo();
112       }
113       return -1;
114  }
```

int ListaAdjAresta::getNumVerticesVizinhos ()

Retorna a quantidade de vértices vizinhos (número de arestas na lista).

Retorna

Número de vértices vizinhos.

```
94                                                     {
95
96       int tamanho = 0;
97       NoAresta* atual = this->cabeca;
98       while (atual != nullptr) {
99            tamanho++;
100            atual = atual->getProximo();
101       }
102       return tamanho;
103  }
```

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

Remove uma aresta específica da lista.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

```
53                                                                          {
54      NoAresta* atual = this->cabeca;
55      NoAresta* anterior = nullptr;
56      while (atual != nullptr) {
57           if (atual->getOrigem() == origem && atual->getDestino() == destino) {
58                if (anterior == nullptr) {
59                     this->cabeca = atual->getProximo();
60                } else {
61                     anterior->setProximo(atual->getProximo());
62                }
63              //cout << "Removida Aresta("<< atual->getIdAresta() <<"): " << origem
<< " -> " << destino << endl;                                              /* { DEBUG }
*/
64                delete atual;
65                grafo->decrementa num arestas grafos();
66                return;
67           }
68           anterior = atual;
69           atual = atual->getProximo();
70      }
71     //cout << "Erro: Aresta " << origem << " -> " << destino << " nao existe." <<
 endl;                                        /* { DEBUG } */
72 }
```

void ListaAdjAresta::remover_primeira_aresta ()

Remove a primeira aresta da lista.

```
75                                                          {
76      if (this->cabeca == nullptr) {
77           cout << "Vertice nao possui arestas" << endl;
78           return;
79      }
80
81      NoAresta* atual = this->cabeca;
82      NoAresta* menor = this->cabeca;
83
84      while (atual != nullptr) {
85           if (atual->getDestino() < menor->getDestino()) {
86                menor = atual;
87           }
88           atual = atual->getProximo();
89      }
90      remover_aresta(menor->getOrigem(), menor->getDestino());
91 }
```

Referência da Classe ListaAdjVertice

Gerencia a lista encadeada de vértices em um grafo.

`#include <ListaAdjVertice.h>`

Membros Públicos

- **ListaAdjVertice** (**GrafoLista** *grafo)

*Construtor da classe **ListaAdjVertice**.*

- **~ListaAdjVertice** ()

*Destrutor da classe **ListaAdjVertice**.*

- **NoVertice** * **getCabeca** ()

Retorna o primeiro nó (cabeça) da lista de vértices.

- int **getNumVertices** ()

Retorna a quantidade total de vértices na lista.

- **NoVertice** * **getVertice** (int id)

Retorna um vértice específico da lista.

- void **adicionar_vertice** (int id, float peso)

Adiciona um vértice à lista.

- void **adicionar_aresta** (int origem, int destino, float peso)

Adiciona uma aresta ao vértice.

- void **remover_aresta** (int origem, int destino)

Remove uma aresta do vértice.

- void **remover_primeira_aresta** (int id)

Remove a primeira aresta do vértice.

- void **remover_vertice** (int id)

Remove um vértice da lista.

- int **getIDAresta** (int origem, int destino)

Retorna o ID da aresta entre dois vértices.

- void **imprimir** ()

Imprime a lista de adjacência dos vértices.

Descrição detalhada

Gerencia a lista encadeada de vértices em um grafo.

Construtores e Destrutores

ListaAdjVertice::ListaAdjVertice (GrafoLista * grafo)

Construtor da classe **ListaAdjVertice**.

Parâmetros

grafo	Ponteiro para o objeto **GrafoLista** que utiliza essa lista.

```
9                                                                   {
10      this->grafo = grafo;
11      this->cabeca = nullptr;
12 }
```

ListaAdjVertice::~ListaAdjVertice ()

Destrutor da classe **ListaAdjVertice**.

```
14                                              {
15      // Desaloca a memoria para cada no da lista de vertices
16      NoVertice* atual = this->cabeca;
17      while (atual != nullptr) {
18           NoVertice* next = atual->getProximo();
19           delete atual;
20           atual = next;
21      }
22 }
```

Documentação das funções

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

Adiciona uma aresta ao vértice.

Parâmetros

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

```
64
           {
65      NoVertice* atual = this->cabeca;
66      while (atual != nullptr) {
67           if (atual->getIdVertice() == origem) {
68                atual->adicionar_aresta(destino, peso);
69           }
```

```
70            atual = atual->getProximo();
71        }
72 }
```

void ListaAdjVertice::adicionar_vertice (int id, float peso)

Adiciona um vértice à lista.

Parâmetros

id	Identificador do novo vértice.
peso	Peso do vértice.

```
54                                                                          {
55        NoVertice* novoNo = new NoVertice(id, peso, grafo);
56        novoNo->setProximo(this->cabeca);
57        this->cabeca = novoNo;
58        grafo->incrementa num vertices grafo();
59
60        //cout << "Adicionado Vertice " << novoNo->getIdVertice() << endl;
                /* { DEBUG } */
61 }
```

NoVertice * ListaAdjVertice::getCabeca ()

Retorna o primeiro nó (cabeça) da lista de vértices.

Retorna

Ponteiro para o nó inicial da lista.

```
26                                                  {
27        return this->cabeca;
28 }
```

int ListaAdjVertice::getIDAresta (int origem, int destino)

Retorna o ID da aresta entre dois vértices.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

Retorna

ID da aresta.

```
158                                                                         {
159       if(this->getVertice(origem) == nullptr) {
160            cout << "Erro: Vertice de origem nao existe!" << endl;
161            return -1;
162       }
163       return getVertice(origem)->getIdAresta(destino);
164 }
```

int ListaAdjVertice::getNumVertices ()

Retorna a quantidade total de vértices na lista.

Retorna

Número de vértices.

```
31                                                  {
32       int tamanho = 0;
33       NoVertice* atual = this->cabeca;
34       while (atual != nullptr) {
35            tamanho++;
36            atual = atual->getProximo();
37       }
38       return tamanho;
39  }
```

NoVertice * ListaAdjVertice::getVertice (int id)

Retorna um vértice específico da lista.

Parâmetros

id	Identificador do vértice.

Retorna

Ponteiro para o nó do vértice.

```
42                                                              {
43       NoVertice* atual = this->cabeca;
44       while (atual != nullptr) {
45            if (atual->getIdVertice() == id) {
46                 return atual;
47            }
48            atual = atual->getProximo();
49       }
50       return nullptr;
51  }
```

void ListaAdjVertice::imprimir ()

Imprime a lista de adjacência dos vértices.

```
167                                          {
168       // Imprime a lista de adjacencia
169     cout << "                                                                    "
<< endl;
170       cout << endl << "--- Lista de Adjacencia ---" << endl << endl ;
171       NoVertice* atual = this->cabeca;
172       while (atual != nullptr) {
173            cout << "Vertice " << atual->getIdVertice() << " -> ";
174            NoAresta* atualAresta = atual->getArestas()->getCabeca();
175            while (atualAresta != nullptr) {
176               cout << atualAresta->getDestino() << "(" << atualAresta->getPeso()
<< ")"<< "   ";
```

```
177                 atualAresta = atualAresta->getProximo();
178            }
179            cout << endl;
180            atual = atual->getProximo();
181       }
182     cout << "_____________________________________________________________________"
<< endl << endl;
183 }
```

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

Remove uma aresta do vértice.

Parâmetros

origem	Identificador do vértice de origem.
destino	Identificador do vértice de destino.

```
75                                                                          {
76       NoVertice* atual = this->cabeca;
77       while (atual != nullptr) {
78            if (atual->getIdVertice() == origem) {
79                 atual->remover_aresta(destino);
80            }
81            atual = atual->getProximo();
82       }
83 }
```

void ListaAdjVertice::remover_primeira_aresta (int id)

Remove a primeira aresta do vértice.

Parâmetros

id	Identificador do vértice.

```
86                                                                  {
87       NoVertice* atual = this->cabeca;
88       while (atual != nullptr) {
89            if (atual->getIdVertice() == id) {
90                 atual->remover_primeira_aresta();
91            }
92            atual = atual->getProximo();
93       }
94 }
```

void ListaAdjVertice::remover_vertice (int id)

Remove um vértice da lista.

Parâmetros

id	Identificador do vértice a ser removido.

```
97                                                          {
98       NoVertice* atual = this->cabeca;
```

```
99      NoVertice* anterior = nullptr;
100      NoVertice* remover =   nullptr;
101      while (atual != nullptr) {
102           if (atual->getIdVertice() == id) {
103                if (anterior == nullptr) {
104                     this->cabeca = atual->getProximo();
105                } else {
106                     anterior->setProximo(atual->getProximo());
107                }
108                remover = atual;
109           }
110           // Remove as arestas que apontam para o vertice a ser removido
111           atual->remover_aresta(id);
112           // Atualiza os ponteiros
113           anterior = atual;
114           atual = atual->getProximo();
115      }
116
117      // Atualiza o contador de arestas
118      if (remover != nullptr) {
119           NoAresta* arestaAtual = remover->getArestas()->getCabeca();
120           while (arestaAtual != nullptr) {
121                grafo->decrementa num arestas grafos();
122                if (!grafo->eh_direcionado()) {
123                     grafo->decrementa_num_arestas_grafos();
124                }
125                arestaAtual = arestaAtual->getProximo();
126           }
127      }
128
129      // Remove o vertice
130      //cout << "Removendo vertice " << remover->getIdVertice() << endl;
     /* { DEBUG } */
131      grafo->decrementa num vertices grafo();
132      delete remover;
133
134      // Recalculando ID dos vertices
135      atual = this->cabeca;
136      while (atual != nullptr) {
137           if (atual->getIdVertice() > id) {
138                atual->setIdVertice(atual->getIdVertice() - 1);
139           }
140           atual = atual->getProximo();
141      }
142
143      // Reorganizando as arestas
144      atual = this->cabeca;
145      while (atual != nullptr) {
146           NoAresta* arestaAtual = atual->getArestas()->getCabeca();
147           while (arestaAtual != nullptr) {
148                if (arestaAtual->getDestino() > id) {
149                  arestaAtual->setVerticeDestino(arestaAtual->getDestino() - 1);
150                }
151                arestaAtual = arestaAtual->getProximo();
152           }
153           atual = atual->getProximo();
154      }
155
156 }
```

Referência da Classe NoAresta

Representa uma aresta em uma lista encadeada.

`#include <NoAresta.h>`

Membros Públicos

- **NoAresta** (int idVerticeOrigem, int idVerticeDestino, float peso, int idAresta)

*Construtor da classe **NoAresta**.*

- **~NoAresta** ()

*Destrutor da classe **NoAresta**.*

- int **getIdAresta** ()

Retorna o identificador da aresta.

- int **getOrigem** ()

Retorna o identificador do vértice de origem.

- int **getDestino** ()

Retorna o identificador do vértice de destino.

- float **getPeso** ()

Retorna o peso da aresta.

- **NoAresta** * **getProximo** ()

Retorna o próximo nó de aresta na lista.

- void **setProximo** (**NoAresta** *proximo)

Define o próximo nó de aresta na lista.

- void **setVerticeOrigem** (int novoId)

Define o identificador do vértice de origem.

- void **setVerticeDestino** (int novoId)

Define o identificador do vértice de destino.

Descrição detalhada

Representa uma aresta em uma lista encadeada.

Construtores e Destrutores

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

Construtor da classe **NoAresta**.

Parâmetros

idVerticeOrigem	Identificador do vértice de origem.
idVerticeDestino	Identificador do vértice de destino.
peso	Peso da aresta.
idAresta	Identificador único da aresta.

```
7
                              {
8       this-> idAresta = idAresta;
9       this->idVerticeOrigem = idVerticeOrigem;
10       this->idVerticeDestino = idVerticeDestino;
11       this->peso = peso;
12       this->proximo = nullptr;
13 }
```

NoAresta::~NoAresta ()

Destrutor da classe **NoAresta**.

```
15                            {
16
17 }
```

Documentação das funções

int NoAresta::getDestino ()

Retorna o identificador do vértice de destino.

Retorna

ID do vértice de destino.

```
31                                  {
32       return idVerticeDestino;
33 }
```

int NoAresta::getIdAresta ()

Retorna o identificador da aresta.

Retorna

ID da aresta.

```
21                                  {
22      return idAresta;
23 }
```

int NoAresta::getOrigem ()

Retorna o identificador do vértice de origem.

Retorna

ID do vértice de origem.

```
26                                  {
27      return idVerticeOrigem;
28 }
```

float NoAresta::getPeso ()

Retorna o peso da aresta.

Retorna

Peso da aresta.

```
36                                  {
37      return peso;
38 }
```

NoAresta * NoAresta::getProximo ()

Retorna o próximo nó de aresta na lista.

Retorna

Ponteiro para o próximo **NoAresta**.

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

void NoAresta::setProximo (NoAresta * proximo)

Define o próximo nó de aresta na lista.

Parâmetros

proximo	Ponteiro para o novo próximo nó.

```
46                                                          {
47      this->proximo = proximo;
48 }
```

void NoAresta::setVerticeDestino (int novoId)

Define o identificador do vértice de destino.

Parâmetros

novoId	Novo identificador para o vértice de destino.

```
56                                                          {
57      this->idVerticeDestino = novoId;
58 }
```

void NoAresta::setVerticeOrigem (int novoId)

Define o identificador do vértice de origem.

Parâmetros

novoId	Novo identificador para o vértice de origem.

```
51                                                         {
52      this->idVerticeOrigem = novoId;
53 }
```

Referência da Classe NoVertice

Representa um vértice em uma lista encadeada com suas arestas associadas.

`#include <NoVertice.h>`

Membros Públicos

- **NoVertice** (int vertice, float peso, **GrafoLista** *grafo)

*Construtor da classe **NoVertice**.*

- **~NoVertice** ()

*Destrutor da classe **NoVertice**.*

- int **getIdVertice** ()

Retorna o identificador do vértice.

- float **getPesoVertice** ()

Retorna o peso do vértice.

- int **getNumArestasVertice** ()

Retorna a quantidade de arestas (grau) do vértice.

- **NoVertice** * **getProximo** ()

Retorna o próximo nó de vértice na lista.

- **ListaAdjAresta** * **getArestas** ()

Retorna a lista encadeada de arestas associadas ao vértice.

- void **setProximo** (**NoVertice** *proximo)

Define o próximo nó de vértice na lista.

- int **setIdVertice** (int novoId)

Define um novo identificador para o vértice.

- int **getNumVizinhos** ()

Retorna o número de vizinhos (grau) do vértice.

- void **adicionar_aresta** (int id, float peso)

Adiciona uma aresta ao vértice.

- void **remover_aresta** (int destino)

Remove uma aresta do vértice.

- void **remover_primeira_aresta** ()

Remove a primeira aresta associada ao vértice.

- int **getIdAresta** (int destino)

Retorna o identificador da aresta que conecta este vértice a outro.

Descrição detalhada

Representa um vértice em uma lista encadeada com suas arestas associadas.

Construtores e Destrutores

NoVertice::NoVertice (int vertice, float peso, GrafoLista * grafo)

Construtor da classe **NoVertice**.

Parâmetros

vertice	Identificador do vértice.
peso	Peso do vértice.
grafo	Ponteiro para o grafo (**GrafoLista**) que contém o vértice.

```
8
 {
9       this->idVertice = idVertice;
10       this->peso = peso;
11       this->proximo = nullptr;
12       this->numArestasVertice = 0;
13       this->grafo = grafo;
14       this->arestas = new ListaAdjAresta(grafo);
15 }
```

NoVertice::~NoVertice ()

Destrutor da classe **NoVertice**.

```
17                              {
18       delete arestas;
19 }
```

Documentação das funções

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

Adiciona uma aresta ao vértice.

Parâmetros

id	Identificador do vértice de destino da nova aresta.
peso	Peso da aresta.

```
64                                                                          {
65      this->arestas->adicionar_aresta(this->idVertice, destino, peso);
66      this->numArestasVertice++;
67 }
```

ListaAdjAresta * NoVertice::getArestas ()

Retorna a lista encadeada de arestas associadas ao vértice.

Retorna

Ponteiro para a **ListaAdjAresta**.

```
49                                                  {
50      return this->arestas;
51 }
```

int NoVertice::getIdAresta (int destino)

Retorna o identificador da aresta que conecta este vértice a outro.

Parâmetros

destino	Identificador do vértice de destino.

Retorna

ID da aresta.

```
81                                                  {
82      return this->arestas->getIdAresta(destino);
83 }
```

int NoVertice::getIdVertice ()

Retorna o identificador do vértice.

Retorna

ID do vértice.

```
23                                      {
24      return this->idVertice;
25 }
```

int NoVertice::getNumArestasVertice ()

Retorna a quantidade de arestas (grau) do vértice.

Retorna

Número de arestas.

```
54                                                  {
55      return this->numArestasVertice;
56 }
```

int NoVertice::getNumVizinhos ()

Retorna o número de vizinhos (grau) do vértice.

Retorna

Número de vizinhos.

```
59                                          {
60      return this->arestas->getNumVerticesVizinhos();
61 }
```

float NoVertice::getPesoVertice ()

Retorna o peso do vértice.

Retorna

Peso do vértice.

```
28                                          {
29      return this->peso;
30 }
```

NoVertice * NoVertice::getProximo ()

Retorna o próximo nó de vértice na lista.

Retorna

Ponteiro para o próximo **NoVertice**.

```
33                                          {
34      return this->proximo;
35 }
```

void NoVertice::remover_aresta (int destino)

Remove uma aresta do vértice.

Parâmetros

destino	Identificador do vértice de destino da aresta a ser removida.

```
70                                                  {
71      this->arestas->remover_aresta(this->idVertice, destino);
```

```
72      this->numArestasVertice--;
73 }
```

void NoVertice::remover_primeira_aresta ()

Remove a primeira aresta associada ao vértice.

```
76                                                  {
77      this->arestas->remover_primeira_aresta();
78      this->numArestasVertice--;
79 }
```

int NoVertice::setIdVertice (int novoId)

Define um novo identificador para o vértice.

Parâmetros

novoId	Novo ID do vértice.

Retorna

O novo identificador atribuído.

```
43                                                  {
44      this->idVertice = novoId;
45      return this->idVertice;
46 }
```

void NoVertice::setProximo (NoVertice * proximo)

Define o próximo nó de vértice na lista.

Parâmetros

proximo	Ponteiro para o novo próximo nó.

```
38                                                      {
39      this->proximo = proximo;
40 }
```

A documentação para essa classe foi gerada a partir dos seguintes arquivos:

C:/Users/fabio/OneDrive/Documentos/Grafos/TrabalhoGrafos/include/**NoVertice.h**
C:/Users/fabio/OneDrive/Documentos/Grafos/TrabalhoGrafos/src/**NoVertice.cpp**

Arquivos

Referência do Arquivo Grafo.h

Declaração da classe base para representação de grafos.

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

Componentes

- class **Grafo***Representa um grafo com propriedades básicas.*

Descrição detalhada

Declaração da classe base para representação de grafos.

Grafo.h

Ir para a documentação desse arquivo.

```
1
5
6 #ifndef GRAFO H
7 #define GRAFO H
8
9 #include "../include/ListaAdjAresta.h"
10 #include "../include/ListaAdjVertice.h"
11 #include <iostream>
12
13 using namespace std;
14
25 class Grafo {
26 protected:
27      int numVertices;
28      int numArestasGrafo;
29      bool direcionado;
30      bool ponderadoVertices;
31      bool ponderadoArestas;
32
33 public:
41      Grafo(int numVertices, bool direcionado, bool ponderadoVertices, bool
ponderadoArestas);
42
46      ~Grafo();
47
48      // Métodos principais
49
54      int n_conexo();
55
60      int get_grau();
61
66      int get_ordem();
67
72      bool eh_direcionado();
73
78      bool vertice ponderado();
79
84      bool aresta ponderada();
85
90      bool eh completo();
91
92      // Função para geração de grafo
93
99      static void carrega_grafo(Grafo* grafo, const string& nomeArquivo);
100
101      // Funções de impressão
102
106      void imprimir_descricao();
107
112      void imprimir_algoritmos_cobertura_vertice(Grafo* grafo);
113
119      void analise algoritmos cobertura vertice(Grafo* grafo, int numVezes);
120
121      // Métodos auxiliares abstratos a serem implementados nas classes derivadas
122
128      virtual ListaAdjAresta* get vizinhos(int id) { return nullptr; };
129
135      virtual int get_num_vizinhos(int id) { return 0; };
136
142      virtual void dfs(int v, bool* visitado){};
143
149      virtual bool existe_vertice(int id)=0;
150
155      int get_num_arestas_grafo();
156
160      void incrementa_num_vertices_grafo();
```

```
161
165       void decrementa_num_vertices_grafo();
166
170       void incrementa num arestas grafos();
171
175       void decrementa num arestas grafos();
176
181       void diminui num arestas grafos(int valor);
182
183       // Métodos abstratos de manipulação de vértices e arestas
184
190       virtual void adicionar_vertice(int id, float peso = 0.0){};
191
198       virtual void adicionar_aresta(int origem, int destino, float peso = 1.0){};
199
204       virtual void remover_primeira_aresta(int id){};
205
210       virtual void remover_vertice(int id){};
211
217       virtual void remover aresta(int origem, int destino){};
218
219       // Métodos abstratos para cálculo de distâncias
220
227       virtual int calcula menor dist(int origem, int destino)=0;
228
233       virtual int calcula maior menor dist();
234
235       // Métodos abstratos para algoritmos de cobertura de vértices
236
241       virtual int alg_guloso_cobertura_vertice() = 0;
242
247       virtual int alg_randomizado_cobertura_vertice() = 0;
248
253       virtual int alg_reativo_cobertura_vertice() = 0;
254 };
255
256 #endif // GRAFO_H
```

Referência do Arquivo C:/Users/fabio/OneDrive/Documentos/Grafos/TrabalhoGrafos/include/GrafoLista.h

Declaração da classe **GrafoLista**, que implementa grafos usando listas de adjacência.

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

Componentes

- class **GrafoLista***Representa um grafo utilizando listas de adjacência.*

Descrição detalhada

Declaração da classe **GrafoLista**, que implementa grafos usando listas de adjacência.

GrafoLista.h

Ir para a documentação desse arquivo.

```
1
5
6 #ifndef GRAFO LISTA H
7 #define GRAFO LISTA H
8
9 #include "Grafo.h"
10 #include "ListaAdjVertice.h"
11 #include "ListaAdjAresta.h"
12
21 class GrafoLista : public Grafo {
22 protected:
23       ListaAdjVertice* listaAdjVertices;
24
25 public:
33       GrafoLista(int numVertices, bool direcionado, bool ponderadoVertices, bool
ponderadoArestas);
34
38       ~GrafoLista();
39
40       // Métodos auxiliares
41
47       int get_num_vizinhos(int id) override;
48
54       void dfs(int id, bool* visitado) override;
55
61       bool existe_vertice(int id) override;
62
63       // Métodos de manipulação de vértices e arestas
64
70       void adicionar vertice(int id, float peso = 0.0) override;
71
78       void adicionar aresta(int origem, int destino, float peso = 1.0) override;
79
84       void remover vertice(int id) override;
85
91       void remover aresta(int origem, int destino) override;
92
97       void remover_primeira_aresta(int id) override;
98
105       int calcula_menor_dist(int origem, int destino);
106
107       // Métodos de impressão
108
112       void imprimeGrafoLista();
113
117       void imprimeListaAdj();
118
119       // Algoritmos gulosos para cobertura de vértices
120
125       int alg guloso cobertura vertice() override;
126
131       int alg randomizado cobertura vertice() override;
132
137       int alg_reativo_cobertura_vertice() override;
138 };
139
140 #endif // GRAFO_LISTA_H
```

Referência do Arquivo C:/Users/fabio/OneDrive/Documentos/Grafos/TrabalhoGrafos/include/GrafoMatriz.h

Declaração da classe **GrafoMatriz**, que implementa grafos usando matriz de adjacência.

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

Componentes

- class **GrafoMatriz***Representa um grafo utilizando uma matriz de adjacência.*

Descrição detalhada

Declaração da classe **GrafoMatriz**, que implementa grafos usando matriz de adjacência.

GrafoMatriz.h

Ir para a documentação desse arquivo.

```
1
5
6 #ifndef GRAFOMATRIZ H
7 #define GRAFOMATRIZ H
8
9 #include "Grafo.h"
10 #include <vector>
11
19 class GrafoMatriz : public Grafo {
20 protected:
21      int** matrizAdj;
22      int tamanhoMatriz;
23
24 public:
32      GrafoMatriz(int numVertices, bool direcionado, bool ponderadoVertices, bool
ponderadoArestas);
33
37      ~GrafoMatriz();
38
39      // Métodos auxiliares
40
46      int grauVertice(int vertice);
47
53      int get num vizinhos(int id) override;
54
60      void dfs(int id, bool* visitado) override;
61
67      bool existe_vertice(int id) override;
68
69      // Métodos de manipulação de vértices e arestas
70
76      void adicionar_vertice(int id, float peso = 0.0) override;
77
84      void adicionar_aresta(int origem, int destino, float peso = 1.0) override;
85
90      void remover vertice(int id) override;
91
97      void remover aresta(int origem, int destino) override;
98
103      void remover primeira aresta(int id) override;
104
111      int calcula_menor_dist(int origem, int destino) override;
112
113      // Métodos de impressão
114
118      void imprimirMatrizAdj();
119
123      void imprimeGrafoMatriz();
124
125      // Algoritmos gulosos para cobertura de vértices
126
131      int alg guloso cobertura vertice() override;
132
137      int alg randomizado cobertura vertice() override;
138
143      int alg reativo cobertura vertice() override;
144
150      int alg_randomizado_cobertura_vertice_com_alpha(double alpha);
151 };
152
153 #endif // GRAFOMATRIZ_H
```

Referência do Arquivo ListaAdjAresta.h

Declaração da classe **ListaAdjAresta**, que gerencia a lista de adjacência para as arestas.
`#include "NoAresta.h"`

Componentes

- class **ListaAdjAresta***Gerencia a lista encadeada de arestas associadas a um vértice.*

Descrição detalhada

Declaração da classe **ListaAdjAresta**, que gerencia a lista de adjacência para as arestas.

ListaAdjAresta.h

Ir para a documentação desse arquivo.

```
1
5
6 #ifndef LISTA ADJ ARESTA H
7 #define LISTA ADJ ARESTA H
8
9 #include "NoAresta.h"
10
11 class GrafoLista;   // Declaração antecipada para evitar dependências circulares
12
17 class ListaAdjAresta {
18 private:
19      NoAresta* cabeca;
20      GrafoLista* grafo;
21
22 public:
27      ListaAdjAresta(GrafoLista* grafo);
28
32      ~ListaAdjAresta();
33
38      NoAresta* getCabeca();
39
44      int getNumVerticesVizinhos();
45
52      void adicionar_aresta(int origem, int destino, float peso);
53
59      void remover_aresta(int origem, int destino);
60
64      void remover primeira aresta();
65
71      int getIdAresta(int destino);
72 };
73
74 #endif // LISTA_ADJ_ARESTA_H
```

Referência do Arquivo ListaAdjVertice.h

Declaração da classe **ListaAdjVertice**, que gerencia a lista de adjacência para os vértices.
`#include "NoVertice.h"`

Componentes

- class **ListaAdjVertice***Gerencia a lista encadeada de vértices em um grafo.*

Descrição detalhada

Declaração da classe **ListaAdjVertice**, que gerencia a lista de adjacência para os vértices.

ListaAdjVertice.h

Ir para a documentação desse arquivo.

```
1
5
6 #ifndef LISTA ADJ VERTICE H
7 #define LISTA ADJ VERTICE H
8
9 #include "NoVertice.h"
10
11 class GrafoLista;   // Declaração antecipada
12
17 class ListaAdjVertice {
18 private:
19      NoVertice* cabeca;
20      GrafoLista* grafo;
21
22 public:
27      ListaAdjVertice(GrafoLista* grafo);
28
32      ~ListaAdjVertice();
33
38      NoVertice* getCabeca();
39
44      int getNumVertices();
45
51      NoVertice* getVertice(int id);
52
58      void adicionar_vertice(int id, float peso);
59
66      void adicionar aresta(int origem, int destino, float peso);
67
73      void remover aresta(int origem, int destino);
74
79      void remover primeira aresta(int id);
80
85      void remover_vertice(int id);
86
93      int getIDAresta(int origem, int destino);
94
98      void imprimir();
99 };
100
101 #endif // LISTA_ADJ_VERTICE_H
```

Referência do Arquivo NoAresta.h

Declaração da classe **NoAresta**, que representa um nó em uma lista de arestas.

Componentes

- class **NoAresta***Representa uma aresta em uma lista encadeada.*

Descrição detalhada

Declaração da classe **NoAresta**, que representa um nó em uma lista de arestas.

NoAresta.h

Ir para a documentação desse arquivo.

```
1
5
6 #ifndef NO_ARESTA_H
7 #define NO_ARESTA_H
8
13 class NoAresta {
14 private:
15      int idAresta;
16      int idVerticeOrigem;
17      int idVerticeDestino;
18      float peso;
19      NoAresta* proximo;
20
21 public:
29      NoAresta(int idVerticeOrigem, int idVerticeDestino, float peso, int idAresta);
30
34      ~NoAresta();
35
40      int getIdAresta();
41
46      int getOrigem();
47
52      int getDestino();
53
58      float getPeso();
59
64      NoAresta* getProximo();
65
70      void setProximo(NoAresta* proximo);
71
76      void setVerticeOrigem(int novoId);
77
82      void setVerticeDestino(int novoId);
83 };
84
85 #endif // NO_ARESTA_H
```

Referência do Arquivo NoVertice.h

Declaração da classe **NoVertice**, que representa um nó em uma lista de vértices.

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

Componentes

- class **NoVertice***Representa um vértice em uma lista encadeada com suas arestas associadas.*

Descrição detalhada

Declaração da classe **NoVertice**, que representa um nó em uma lista de vértices.

NoVertice.h

Ir para a documentação desse arquivo.

```
1
5
6 #ifndef NO_VERTICE_H
7 #define NO_VERTICE_H
8
9 #include "ListaAdjAresta.h"
10 #include "NoAresta.h"
11
12 class GrafoLista;   // Declaração antecipada
13
18 class NoVertice {
19 private:
20      int idVertice;
21      float peso;
22      int numArestasVertice;
23      NoVertice* proximo;
24      ListaAdjAresta* arestas;
25      GrafoLista* grafo;
26
27 public:
34      NoVertice(int vertice, float peso, GrafoLista* grafo);
35
39      ~NoVertice();
40
41      // Getters e Setters
42
47      int getIdVertice();
48
53      float getPesoVertice();
54
59      int getNumArestasVertice();
60
65      NoVertice* getProximo();
66
71      ListaAdjAresta* getArestas();
72
77      void setProximo(NoVertice* proximo);
78
84      int setIdVertice(int novoId);
85
86      // Métodos auxiliares
87
92      int getNumVizinhos();
93
99      void adicionar_aresta(int id, float peso);
100
105      void remover aresta(int destino);
106
110      void remover primeira aresta();
111
117      int getIdAresta(int destino);
118 };
119
120 #endif // NO_VERTICE_H
```

Referência do Arquivo C:/Users/fabio/OneDrive/Documentos/Grafos/TrabalhoGrafos/main.cpp

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

Funções

- void **comando_invalido** ()
- int **main** (int argc, char *argv[])

Funções

void comando_invalido ()

```
21                                  {
22       cerr << endl << "Comando invalido. Use: " << endl;
23      cerr << "$ ./main -d -m grafo.txt: Para imprimir a descricao do Grafo Matriz"
<< endl;
24       cerr << "$ ./main -d -l grafo.txt: Para imprimir a descricao do Grafo Lista"
<< endl;
25     cerr << "$ ./main -p -m grafo.txt: Para imprimir a solucao de cobertura de vertices
do Grafo Matriz" << endl;
26     cerr << "$ ./main -p -l grafo.txt: Para imprimir a solucao de cobertura de vertices
do Grafo Lista" << endl <<endl;
27 }
```

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

```
29                                                  {
30       if (argc != 4) {
31            comando_invalido();
32            return 1;
33       }
34       string modo = argv[2];
35       string arquivo = "./entradas/" + string(argv[3]);
36
37       if (modo == "-m") {
38            cout << endl << "============================== MATRIZ
==============================" << endl << endl;
39            GrafoMatriz grafoMatriz(0, false, false, false);
40            grafoMatriz.carrega_grafo(&grafoMatriz, arquivo);
41
42            if (string(argv[1]) == "-d") {
43                 grafoMatriz.imprimir_descricao();
44            } else if (string(argv[1]) == "-p") {
45                 grafoMatriz.imprimir_algoritmos_cobertura_vertice(&grafoMatriz);
46            } else if(string(argv[1]) == "-a"){
47              grafoMatriz.analise algoritmos cobertura vertice(&grafoMatriz, 100);
48            }
49            else{
50                 comando invalido();
51                 return 1;
```

```
52            }
53            cout << endl << "============================== FIM MATRIZ
==============================" << endl << endl;
54       } else if (modo == "-l") {
55            cout << endl << "============================== LISTA
==============================" << endl << endl;
56            GrafoLista grafoLista(0, false, false, false);
57            grafoLista.carrega grafo(&grafoLista, arquivo);
58
59            if (string(argv[1]) == "-d") {
60                 grafoLista.imprimir_descricao();
61            } else if (string(argv[1]) == "-p") {
62                 grafoLista.imprimir algoritmos cobertura vertice(&grafoLista);
63            } else if(string(argv[1]) == "-a"){
64                grafoLista.analise_algoritmos_cobertura_vertice(&grafoLista, 100);
65            } else{
66                 comando_invalido();
67                 return 1;
68            }
69            cout << endl <<   "============================== FIM LISTA
==============================" << endl << endl;
70       } else {
71            comando_invalido();
72            return 1;
73       }
74       return 0;
75  }
```

Referência do Arquivo Grafo.cpp

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

Referência do Arquivo GrafoLista.cpp

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

Referência do Arquivo GrafoMatriz.cpp

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

Referência do Arquivo ListaAdjAresta.cpp

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

Referência do Arquivo ListaAdjVertice.cpp

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

Referência do Arquivo NoAresta.cpp

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

Referência do Arquivo NoVertice.cpp

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