Estruturas de Dados Avançadas (EDA)

Generated by Doxygen 1.11.0

Chapter 1

Fase 2: Grafos

1.0.0.1 Instituto Politécnico do Cávado e do Ave (IPCA) - Barcelos

Aluno: Hugo Cruz (a23010)

1.0.1 Introdução

Este projeto em C tem como objetivo calcular o somatório máximo possível de inteiros a partir de uma matriz de dimensões arbitrárias, considerando regras específicas de conexão entre os inteiros. Para isso, aplicamos conceitos avançados de teoria dos grafos e programação em C, utilizando estruturas de dados e algoritmos de procura.

1.0.2 Estrutura do Projeto

- **doc/:** Documentação gerada pelo Doxygen.
- **src/:** Código fonte do projeto, incluindo uma biblioteca estática.
  - **Grafos/:** Código fonte utilizado para gerar (`Grafos.lib`)
  - **Main/:** Contém o código principal.
    - **libs/:** Biblioteca estática (`Grafos.lib`).

1.0.3 Funcionalidades Implementadas

1. **Estrutura de Dados GR:** Definição de uma estrutura de dados GR para representar um grafo orientado capaz de suportar um número de vértices de forma dinâmica, incluindo funções básicas para a manipulação do grafo.
2. **Modelagem do Problema com Grafos:** Modelagem do problema utilizando grafos, onde cada elemento da matriz é representado por um vértice e as arestas são representadas por valores lidos.
3. **Carregamento de Dados:** Carregamento dos dados de uma matriz de inteiros a partir de um arquivo de texto, permitindo grafos de qualquer dimensão.
4. **Operações de Manipulação de Grafos:** Implementação de operações de manipulação de grafos, incluindo procura em profundidade ou em largura, para identificar todos os caminhos possíveis que atendem às regras de conexão definidas, e cálculo da soma dos valores dos vértices em um dado caminho.
5. **Encontrar o Caminho com a Menor Soma:** Utilização das estruturas e algoritmos desenvolvidos para encontrar o caminho que proporciona a maimenoror soma possível dos inteiros na estrutura GR, seguindo a regra de conexão estabelecida, fornecendo tanto a soma total quanto o caminho correspondente.

1.0.4 Autores

- `Hugo Cruz (@hugoc03)`

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

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

Chapter 4

Data Structure Documentation

4.1 Adjacente Struct Reference

Estrutura de uma adjacência num grafo.
`#include <adjacente.h>`

Data Fields

- int **id**

  Identificador único da adjacência.
- int **peso**

  Peso associado à adjacência.
- struct Adjacente * **next**

  Apontador para a próxima adjacência na lista de adjacências.

4.1.1 Detailed Description

Estrutura de uma adjacência num grafo.
Esta estrutura representa uma adjacência num grafo. Cada adjacência tem um identificador único, um peso associado à adjacência e um apontador para a próxima adjacência na lista de adjacências (next).
The documentation for this struct was generated from the following files:

- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Grafos/adjacente.h
- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Main/libs/adjacente.h

4.2 AdjacenteFile Struct Reference

Estrutura de uma adjacência utilizada para armazenar adjacências em um ficheiro binário.
`#include <adjacente.h>`

Data Fields

- int **id**

  Identificador único da adjacência.
- int **peso**

  Peso associado à adjacência.

4.2.1 Detailed Description

Estrutura de uma adjacência utilizada para armazenar adjacências em um ficheiro binário.
The documentation for this struct was generated from the following files:

- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Grafos/adjacente.h
- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Main/libs/adjacente.h

4.3 Grafo Struct Reference

Estrutura de dados para um Grafo.
`#include <grafo.h>`

Data Fields

- Vertice * **inicioGrafo**

  Apontador para o primeiro vértice do grafo.

4.3.1 Detailed Description

Estrutura de dados para um Grafo.
A estrutura Grafo é uma representação de um grafo em memória, onde inicioGrafo é um apontador para o primeiro vértice do grafo.
The documentation for this struct was generated from the following files:

- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Grafos/grafo.h
- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Main/libs/grafo.h

4.4 Vertice Struct Reference

Estrutura de um vértice num grafo.
`#include <vertices.h>`

Data Fields

- int **id**

  Identificador único do vértice.
- struct Vertice * **nextV**

  Apontador para o próximo vértice na lista de vértices.
- Adjacente * **nextA**

  Apontador para o primeiro adjacente na lista de adjacências.

4.4.1 Detailed Description

Estrutura de um vértice num grafo.
Esta estrutura representa um vértice num grafo. Cada vértice tem um identificador único (id), um apontador para o próximo vértice na lista de vértices (nextV), um apontador para o primeiro adjacente na lista de adjacências (nextA) e um indicador se o vértice foi visitado ou não (visitado).
The documentation for this struct was generated from the following files:

- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Grafos/vertices.h
- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Main/libs/vertices.h

4.5 VerticeFile Struct Reference

Estrutura de um vértice utilizada para armazenar vértices em um ficheiro binário.
`#include <vertices.h>`

Data Fields

- int **id**

  Identificador único do vértice.

4.5.1 Detailed Description

Estrutura de um vértice utilizada para armazenar vértices em um ficheiro binário.
The documentation for this struct was generated from the following files:

- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Grafos/vertices.h
- C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Main/libs/vertices.h

Chapter 5

File Documentation

5.1 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/adjacente.c File Reference

Implementação de funções para manipular listas de adjacências.

```
#include "adjacente.h"
```

Functions

- void ApagarAdjacencia (Adjacente *a)

  Liberta a memória alocada para a lista.
- Adjacente * CriarAdjacencia (int idDestino, int peso, bool *inf)

  Cria uma nova adjacência.
- Adjacente * InserirAdjacenciaLista (Adjacente *inicio, int idDestino, int peso, bool *inf)

  Insere e cria uma nova adjacência e coloca a no final de uma lista.
- Adjacente * EliminaUmaAdj (Adjacente *inicio, int idDestino, bool *inf)

  Elimina uma adjacência da lista de adjacências.
- Adjacente * ElimanaTodasAdj (Adjacente *adj, bool *inf)

  Apaga todas as adjacências de uma lista.

5.1.1 Detailed Description

Implementação de funções para manipular listas de adjacências.

Author

Hugo Cruz (a23010)

Version

93.0

Date

2024-05-15

Copyright

5.1.2 Function Documentation

5.1.2.1 ApagarAdjacencia()

```
void ApagarAdjacencia (
            Adjacente * a)
```

Liberta a memória alocada para a lista.
Esta função liberta a memória alocada para a lista de adjacências.

Parameters

a	Apontador para a lista a eliminar

```
00021 {
00022     if (a != NULL)
00023     {
00024         free(a); //Liberta a memória alocada
00025         a = NULL;
00026     }
00027
00028 }
```

5.1.2.2 CriarAdjacencia()

```
Adjacente * CriarAdjacencia (
            int idDestino,
            int peso,
            bool * inf)
```

Cria uma nova adjacência.
Esta função cria uma nova adjacência com um identificador de destino e um peso. Também inicializa os apontadores para a proxima adjacência

Parameters

idDestino	Identificador do vértice de destino de uma adjacências.
peso	Distância entre adjacências.
inf	Apontador para o estado da funcionalidade.

Returns

Apontador para uma nova adjacência.

```
00042 {
00043     *inf = false;
00044
00045     Adjacente* aux = (Adjacente*)malloc(sizeof(Adjacente)); // Aloca memória para a estrura Adjacente
00046     if (aux == NULL) return NULL;
00047
00048     aux->id = idDestino; //Atribui valor a nova adjacência
00049     aux->peso = peso;
00050     aux->next = NULL; //Inicializa o apontador para next com um valor nulo
00051
00052     *inf = true;
00053     return aux;
00054 }
```

5.1.2.3 ElimanaTodasAdj()

```
Adjacente * ElimanaTodasAdj (
            Adjacente * adj,
            bool * inf)
```

Apaga todas as adjacências de uma lista.
Esta função remove todas as adjacências de uma lista.

Parameters

adj	Apontador para o início da lista de adjacências.
inf	Apontador para um bool que indica se a execução foi bem sucedida.

Returns

NULL após apagar todas as adjacências.

```
00160 {
00161     *inf = false;
00162
00163     if (adj == NULL) return NULL;
00164
00165     Adjacente* aux = adj;
00166
00167     //Corre a lista de adjacências até ao fim e apaga todas as adjacências.
00168     while (aux)
00169     {
00170         if (aux)
00171         {
00172             adj = aux->next;
00173         }
00174
00175         ApagarAdjacencia(aux);
00176         aux = adj;
00177     }
00178
00179     adj = NULL;
00180     *inf = true;
00181     return adj;
00182 }
```

5.1.2.4 EliminaUmaAdj()

```
Adjacente * EliminaUmaAdj (
            Adjacente * inicio,
            int idDestino,
            bool * inf)
```

Elimina uma adjacência da lista de adjacências.
Esta função remove uma adjacência específica da lista de adjacências.

Parameters

inicio	Apontador para o início de uma lista de adjacências.
idDestino	Identificador do destino da adjacência a eliminar.
inf	Apontador para um bool que indica se a execução foi bem sucedida.

Returns

Apontador para o início da lista de adjacências.

```
00112 {
00113     *inf = false;
00114
00115     if (inicio == NULL)
00116     {
00117         return NULL;
00118     }
00119
00120     Adjacente* aux = inicio;//Guarda a adjacências atual
00121     Adjacente* ant = NULL; //Guarda a adjacências anterior
00122
00123     while (aux && aux->id != idDestino) // Anda até encontar o destino ou o fim da lista
00124     {
00125         ant = aux;
00126         aux = aux->next;
00127     }
00128
00129     //Não encontrou o vertice a ser removido
00130     if (aux == NULL)
00131     {
00132         *inf = false;
```

```
00133         return inicio;
00134     }
00135
00136     if (ant == NULL) //Insere no topo da lista
00137     {
00138         inicio = aux->next;
00139     }
00140     else
00141     {
00142         ant->next = aux->next;
00143     }
00144
00145     ApagarAdjacencia(aux);
00146     *inf = true;
00147     return inicio;
00148 }
```

5.1.2.5 InserirAdjacenciaLista()

```
Adjacente * InserirAdjacenciaLista (
            Adjacente * inicio,
            int idDestino,
            int peso,
            bool * inf)
```

Insere e cria uma nova adjacência e coloca a no final de uma lista.
Esta função cria e insere uma nova adjacência com o identificador do vértice de destino e a distância entre as adjacências no final de uma lista de adjacências.

Parameters

inicio	Aponta para o início da lista de adjacências.
idDestino	Identificador do vértice de destino da adjacência.
peso	Distância entre as adjacências.
inf	Apontador para o estado da funcionalidade.

Returns

Aponta para o início da lista de adjacências após a inserção.

```
00070 {
00071     *inf = false;
00072
00073     Adjacente* adj = CriarAdjacencia(idDestino, peso, inf); //Cria uma adjacências
00074
00075     if (adj == NULL)
00076     {
00077         perror("CriarAdjacencia");
00078         return NULL;
00079     }
00080
00081     if (inicio == NULL)
00082     {
00083         inicio = adj;
00084     }
00085     else
00086     {
00087         Adjacente* aux = inicio;
00088
00089         while (aux->next != NULL) //Coloca-se no fim da lista
00090         {
00091             aux = aux->next;
00092         }
00093
00094         aux->next = adj; //Adiciona a lista
00095     }
00096
00097     *inf = true;
00098     return inicio;
00099 }
```

5.2 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/adjacente.h File Reference

Este ficheiro contém as definições das estruturas de dados para as adjacências num grafo.

```
#include <stdbool.h>
#include <stdlib.h>
```

Data Structures

- struct Adjacente

  Estrutura de uma adjacência num grafo.
- struct AdjacenteFile

  Estrutura de uma adjacência utilizada para armazenar adjacências em um ficheiro binário.

Typedefs

- typedef struct Adjacente Adjacente

  Estrutura de uma adjacência num grafo.
- typedef struct AdjacenteFile **AdjacenteFile**

  Estrutura de uma adjacência utilizada para armazenar adjacências em um ficheiro binário.

Functions

- void ApagarAdjacencia (Adjacente ∗a)

  Liberta a memória alocada para a lista.
- Adjacente ∗ CriarAdjacencia (int idDestino, int peso, bool ∗inf)

  Cria uma nova adjacência.
- Adjacente ∗ InserirAdjacenciaLista (Adjacente ∗inicio, int idDestino, int peso, bool ∗inf)

  Insere e cria uma nova adjacência e coloca a no final de uma lista.
- Adjacente ∗ EliminaUmaAdj (Adjacente ∗inicio, int idDestino, bool ∗inf)

  Elimina uma adjacência da lista de adjacências.
- Adjacente ∗ ElimanaTodasAdj (Adjacente ∗adj, bool ∗inf)

  Apaga todas as adjacências de uma lista.

5.2.1 Detailed Description

Este ficheiro contém as definições das estruturas de dados para as adjacências num grafo.

Author

Hugo Cruz (a23010)

Version

41.0

Date

2024-05-15

Copyright

Copyright (c) 2024

5.2.2 Typedef Documentation

5.2.2.1 Adjacente

```
typedef struct Adjacente Adjacente
```

Estrutura de uma adjacência num grafo.
Esta estrutura representa uma adjacência num grafo. Cada adjacência tem um identificador único, um peso associado à adjacência e um apontador para a próxima adjacência na lista de adjacências (next).

5.2.3 Function Documentation

5.2.3.1 ApagarAdjacencia()

```
void ApagarAdjacencia (
            Adjacente * a)
```

Liberta a memória alocada para a lista.
Esta função liberta a memória alocada para a lista de adjacências.

Parameters

a	Apontador para a lista a eliminar

```
00021 {
00022     if (a != NULL)
00023     {
00024         free(a); //Liberta a memória alocada
00025         a = NULL;
00026     }
00027
00028 }
```

5.2.3.2 CriarAdjacencia()

```
Adjacente * CriarAdjacencia (
            int idDestino,
            int peso,
            bool * inf)
```

Cria uma nova adjacência.
Esta função cria uma nova adjacência com um identificador de destino e um peso. Também inicializa os apontadores para a proxima adjacência

Parameters

idDestino	Identificador do vértice de destino de uma adjacências.
peso	Distância entre adjacências.
inf	Apontador para o estado da funcionalidade.

Returns

Apontador para uma nova adjacência.

```
00042 {
00043     *inf = false;
00044
00045     Adjacente* aux = (Adjacente*)malloc(sizeof(Adjacente)); // Aloca memória para a estrura Adjacente
00046     if (aux == NULL) return NULL;
00047
00048     aux->id = idDestino; //Atribui valor a nova adjacência
00049     aux->peso = peso;
00050     aux->next = NULL; //Inicializa o apontador para next com um valor nulo
00051
00052     *inf = true;
00053     return aux;
00054 }
```

5.2.3.3 ElimanaTodasAdj()

```
Adjacente * ElimanaTodasAdj (
            Adjacente * adj,
            bool * inf)
```

Apaga todas as adjacências de uma lista.
Esta função remove todas as adjacências de uma lista.

Parameters

adj	Apontador para o início da lista de adjacências.
inf	Apontador para um bool que indica se a execução foi bem sucedida.

Returns

NULL após apagar todas as adjacências.

```
00160 {
00161     *inf = false;
00162
00163     if (adj == NULL) return NULL;
00164
00165     Adjacente* aux = adj;
00166
00167     //Corre a lista de adjacências até ao fim e apaga todas as adjacências.
00168     while (aux)
00169     {
00170         if (aux)
00171         {
00172             adj = aux->next;
00173         }
00174
00175         ApagarAdjacencia(aux);
00176         aux = adj;
00177     }
00178
00179     adj = NULL;
00180     *inf = true;
00181     return adj;
00182 }
```

5.2.3.4 EliminaUmaAdj()

```
Adjacente * EliminaUmaAdj (
            Adjacente * inicio,
            int idDestino,
            bool * inf)
```

Elimina uma adjacência da lista de adjacências.
Esta função remove uma adjacência específica da lista de adjacências.

Parameters

inicio	Apontador para o início de uma lista de adjacências.
idDestino	Identificador do destino da adjacência a eliminar.
inf	Apontador para um bool que indica se a execução foi bem sucedida.

Returns

Apontador para o início da lista de adjacências.

```
00112 {
00113     *inf = false;
00114
00115     if (inicio == NULL)
00116     {
00117         return NULL;
00118     }
00119
00120     Adjacente* aux = inicio;//Guarda a adjacências atual
```

```
00121     Adjacente* ant = NULL; //Guarda a adjacências anterior
00122
00123     while (aux && aux->id != idDestino) // Anda até encontar o destino ou o fim da lista
00124     {
00125         ant = aux;
00126         aux = aux->next;
00127     }
00128
00129     //Não encontrou o vertice a ser removido
00130     if (aux == NULL)
00131     {
00132         *inf = false;
00133         return inicio;
00134     }
00135
00136     if (ant == NULL) //Insere no topo da lista
00137     {
00138         inicio = aux->next;
00139     }
00140     else
00141     {
00142         ant->next = aux->next;
00143     }
00144
00145     ApagarAdjacencia(aux);
00146     *inf = true;
00147     return inicio;
00148 }
```

5.2.3.5 InserirAdjacenciaLista()

```
Adjacente * InserirAdjacenciaLista (
            Adjacente * inicio,
            int idDestino,
            int peso,
            bool * inf)
```

Insere e cria uma nova adjacência e coloca a no final de uma lista.
Esta função cria e insere uma nova adjacência com o identificador do vértice de destino e a distância entre as adjacências no final de uma lista de adjacências.

Parameters

inicio	Aponta para o início da lista de adjacências.
idDestino	Identificador do vértice de destino da adjacência.
peso	Distância entre as adjacências.
inf	Apontador para o estado da funcionalidade.

Returns

Aponta para o início da lista de adjacências após a inserção.

```
00070 {
00071     *inf = false;
00072
00073     Adjacente* adj = CriarAdjacencia(idDestino, peso, inf); //Cria uma adjacências
00074
00075     if (adj == NULL)
00076     {
00077         perror("CriarAdjacencia");
00078         return NULL;
00079     }
00080
00081     if (inicio == NULL)
00082     {
00083         inicio = adj;
00084     }
00085     else
00086     {
00087         Adjacente* aux = inicio;
00088
00089         while (aux->next != NULL) //Coloca-se no fim da lista
00090         {
00091             aux = aux->next;
00092         }
00093
```

```
00094         aux->next = adj; //Adiciona a lista
00095     }
00096
00097     *inf = true;
00098     return inicio;
00099 }
```

5.3 adjacente.h

Go to the documentation of this file.

```
00001
00012 #ifndef ADJACENTE_H
00013 #define ADJACENTE_H
00014
00015 #include <stdbool.h>
00016 #include <stdlib.h>
00017
00024 typedef struct Adjacente
00025 {
00026     int id;
00027     int peso;
00028     struct Adjacente *next;
00029
00030 } Adjacente;
00031
00036 typedef struct AdjacenteFile
00037 {
00038     int id;
00039     int peso;
00040
00041 } AdjacenteFile;
00042
00050 void ApagarAdjacencia(Adjacente *a);
00051
00063 Adjacente *CriarAdjacencia(int idDestino, int peso, bool *inf); // Cria uma adjcencia
00064
00077 Adjacente *InserirAdjacenciaLista(Adjacente *inicio, int idDestino, int peso, bool *inf); // Cria e
      coloca na lista de adjacencias
00078
00089 Adjacente *EliminaUmaAdj(Adjacente *inicio, int idDestino, bool *inf); // Apaga uma adjacencia
00090
00100 Adjacente *ElimanaTodasAdj(Adjacente *adj, bool *inf); // Apaga listas de adjacencia toda
00101
00102 #endif
```

5.4 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Main/libs/adjacente.h File Reference

Este ficheiro contém as definições das estruturas de dados para as adjacências num grafo.

```
#include <stdbool.h>
#include <stdlib.h>
```

Data Structures

- struct Adjacente

  Estrutura de uma adjacência num grafo.
- struct AdjacenteFile

  Estrutura de uma adjacência utilizada para armazenar adjacências em um ficheiro binário.

Typedefs

- typedef struct Adjacente Adjacente

  Estrutura de uma adjacência num grafo.
- typedef struct AdjacenteFile **AdjacenteFile**

  Estrutura de uma adjacência utilizada para armazenar adjacências em um ficheiro binário.

Functions

- void ApagarAdjacencia (Adjacente ∗a)

  Liberta a memória alocada para a lista.
- Adjacente ∗ CriarAdjacencia (int idDestino, int peso, bool ∗inf)

  Cria uma nova adjacência.
- Adjacente ∗ InserirAdjacenciaLista (Adjacente ∗inicio, int idDestino, int peso, bool ∗inf)

  Insere e cria uma nova adjacência e coloca a no final de uma lista.
- Adjacente ∗ EliminaUmaAdj (Adjacente ∗inicio, int idDestino, bool ∗inf)

  Elimina uma adjacência da lista de adjacências.
- Adjacente ∗ ElimanaTodasAdj (Adjacente ∗adj, bool ∗inf)

  Apaga todas as adjacências de uma lista.

5.4.1 Detailed Description

Este ficheiro contém as definições das estruturas de dados para as adjacências num grafo.

Author

Hugo Cruz (a23010)

Version

41.0

Date

2024-05-15

Copyright

Copyright (c) 2024

5.4.2 Typedef Documentation

5.4.2.1 Adjacente

```
typedef struct Adjacente Adjacente
```

Estrutura de uma adjacência num grafo.
Esta estrutura representa uma adjacência num grafo. Cada adjacência tem um identificador único, um peso associado à adjacência e um apontador para a próxima adjacência na lista de adjacências (next).

5.4.3 Function Documentation

5.4.3.1 ApagarAdjacencia()

```
void ApagarAdjacencia (
            Adjacente * a)
```

Liberta a memória alocada para a lista.
Esta função liberta a memória alocada para a lista de adjacências.

Parameters

a	Apontador para a lista a eliminar

```
00021 {
00022     if (a != NULL)
00023     {
00024         free(a); //Liberta a memória alocada
00025         a = NULL;
00026     }
00027
00028 }
```

5.4.3.2 CriarAdjacencia()

```
Adjacente * CriarAdjacencia (
            int idDestino,
            int peso,
            bool * inf)
```

Cria uma nova adjacência.
Esta função cria uma nova adjacência com um identificador de destino e um peso. Também inicializa os apontadores para a proxima adjacência

Parameters

idDestino	Identificador do vértice de destino de uma adjacências.
peso	Distância entre adjacências.
inf	Apontador para o estado da funcionalidade.

Returns

Apontador para uma nova adjacência.

```
00042 {
00043     *inf = false;
00044
00045     Adjacente* aux = (Adjacente*)malloc(sizeof(Adjacente)); // Aloca memória para a estrura Adjacente
00046     if (aux == NULL) return NULL;
00047
00048     aux->id = idDestino; //Atribui valor a nova adjacência
00049     aux->peso = peso;
00050     aux->next = NULL; //Inicializa o apontador para next com um valor nulo
00051
00052     *inf = true;
00053     return aux;
00054 }
```

5.4.3.3 ElimanaTodasAdj()

```
Adjacente * ElimanaTodasAdj (
            Adjacente * adj,
            bool * inf)
```

Apaga todas as adjacências de uma lista.
Esta função remove todas as adjacências de uma lista.

Parameters

adj	Apontador para o início da lista de adjacências.
inf	Apontador para um bool que indica se a execução foi bem sucedida.

Returns

NULL após apagar todas as adjacências.

```
00160 {
00161     *inf = false;
00162
00163     if (adj == NULL) return NULL;
00164
00165     Adjacente* aux = adj;
00166
00167     //Corre a lista de adjacências até ao fim e apaga todas as adjacências.
00168     while (aux)
00169     {
00170         if (aux)
00171         {
00172             adj = aux->next;
00173         }
00174
00175         ApagarAdjacencia(aux);
00176         aux = adj;
00177     }
```

```
00178
00179     adj = NULL;
00180     *inf = true;
00181     return adj;
00182 }
```

5.4.3.4 EliminaUmaAdj()

```
Adjacente * EliminaUmaAdj (
            Adjacente * inicio,
            int idDestino,
            bool * inf)
```

Elimina uma adjacência da lista de adjacências.
Esta função remove uma adjacência específica da lista de adjacências.

Parameters

inicio	Apontador para o início de uma lista de adjacências.
idDestino	Identificador do destino da adjacência a eliminar.
inf	Apontador para um bool que indica se a execução foi bem sucedida.

Returns

Apontador para o início da lista de adjacências.

```
00112 {
00113     *inf = false;
00114
00115     if (inicio == NULL)
00116     {
00117         return NULL;
00118     }
00119
00120     Adjacente* aux = inicio;//Guarda a adjacências atual
00121     Adjacente* ant = NULL; //Guarda a adjacências anterior
00122
00123     while (aux && aux->id != idDestino) // Anda até encontar o destino ou o fim da lista
00124     {
00125         ant = aux;
00126         aux = aux->next;
00127     }
00128
00129     //Não encontrou o vertice a ser removido
00130     if (aux == NULL)
00131     {
00132         *inf = false;
00133         return inicio;
00134     }
00135
00136     if (ant == NULL) //Insere no topo da lista
00137     {
00138         inicio = aux->next;
00139     }
00140     else
00141     {
00142         ant->next = aux->next;
00143     }
00144
00145     ApagarAdjacencia(aux);
00146     *inf = true;
00147     return inicio;
00148 }
```

5.4.3.5 InserirAdjacenciaLista()

```
Adjacente * InserirAdjacenciaLista (
            Adjacente * inicio,
            int idDestino,
            int peso,
            bool * inf)
```

Insere e cria uma nova adjacência e coloca a no final de uma lista.

Esta função cria e insere uma nova adjacência com o identificador do vértice de destino e a distância entre as adjacências no final de uma lista de adjacências.

Parameters

inicio	Aponta para o início da lista de adjacências.
idDestino	Identificador do vértice de destino da adjacência.
peso	Distância entre as adjacências.
inf	Apontador para o estado da funcionalidade.

Returns

Aponta para o início da lista de adjacências após a inserção.

```
00070 {
00071     *inf = false;
00072 
00073     Adjacente* adj = CriarAdjacencia(idDestino, peso, inf); //Cria uma adjacências
00074 
00075     if (adj == NULL)
00076     {
00077         perror("CriarAdjacencia");
00078         return NULL;
00079     }
00080 
00081     if (inicio == NULL)
00082     {
00083         inicio = adj;
00084     }
00085     else
00086     {
00087         Adjacente* aux = inicio;
00088 
00089         while (aux->next != NULL) //Coloca-se no fim da lista
00090         {
00091             aux = aux->next;
00092         }
00093 
00094         aux->next = adj; //Adiciona a lista
00095     }
00096 
00097     *inf = true;
00098     return inicio;
00099 }
```

5.5 adjacente.h

Go to the documentation of this file.

```
00001 
00012 #ifndef ADJACENTE_H
00013 #define ADJACENTE_H
00014 
00015 #include <stdbool.h>
00016 #include <stdlib.h>
00017 
00024 typedef struct Adjacente
00025 {
00026     int id;
00027     int peso;
00028     struct Adjacente *next;
00029 
00030 } Adjacente;
00031 
00036 typedef struct AdjacenteFile
00037 {
00038     int id;
00039     int peso;
00040 
00041 } AdjacenteFile;
00042 
00050 void ApagarAdjacencia(Adjacente *a);
00051 
00063 Adjacente *CriarAdjacencia(int idDestino, int peso, bool *inf); // Cria uma adjcencia
00064 
00077 Adjacente *InserirAdjacenciaLista(Adjacente *inicio, int idDestino, int peso, bool *inf); // Cria e
      coloca na lista de adjacencias
00078 
00089 Adjacente *EliminaUmaAdj(Adjacente *inicio, int idDestino, bool *inf); // Apaga uma adjacencia
00090 
00100 Adjacente *ElimanaTodasAdj(Adjacente *adj, bool *inf); // Apaga listas de adjacencia toda
00101 
00102 #endif
```

5.6 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/caminhos.c File Reference

Este ficheiro contém funções para manipular caminhos num grafo.
`#include "caminhos.h"`

Functions

- void InicializarArrays (int *dis[ ])
  Inicializa os arrays de distâncias.
- int ContadorVertices (Grafo *g)
  Conta o número de vértices num grafo.
- int DistanciaMinima (int distancia[ ], bool visitado[ ], int n)
  Encontra a o vértice com a distância mínima.
- void Dijkstra (Grafo *g, int origem, int distanciasFinais[ ], int verticeAnt[ ])
  Algoritmo de Dijkstra.
- int DistanciaMinimaEntreVertices (Grafo *g, int origem, int destino)
  Calcula a distância mínima entre dois vértices num grafo.
- bool ExisteCaminhoGrafo (Grafo *g, int origem, int destino)
  Verifica se existe um caminho entre dois vértices num grafo.
- Grafo * CriarGrafoCaminhoMaisCurto (Grafo *g)
  Cria um grafo com os caminhos mais curto para cada vertice.

5.6.1 Detailed Description

Este ficheiro contém funções para manipular caminhos num grafo.

Author

Hugo Cruz (a23010)

Version

210.1

Date

2024-05-19

Copyright

Copyright (c) 2024

5.6.2 Function Documentation

5.6.2.1 ContadorVertices()

```
int ContadorVertices (
            Grafo * g)
```

Conta o número de vértices num grafo.
Esta função conta o número de vértices num grafo. Retorna o número de vértices no grafo.

Parameters

g	O grafo a ser analisado.

Returns

int O número de vértices no grafo.

```
00044 {
00045     if (g == NULL)
00046     {
00047         return 0;
00048     }
00049 
00050     int contador = 0;
00051 
00052     Vertice* aux = g->inicioGrafo;
00053 
00054     //Avança na lista e conta o número de vertices.
00055     while (aux)
00056     {
00057         contador++;
00058         aux = aux->nextV;
00059     }
00060 
00061     return contador;
00062 }
```

5.6.2.2 CriarGrafoCaminhoMaisCurto()

```
Grafo * CriarGrafoCaminhoMaisCurto (
            Grafo * g)
```

Cria um grafo com os caminhos mais curto para cada vertice.
Esta função cria um novo grafo com os caminhos mais curtos para cada vértice

Parameters

g	Apontador para o grafo original

Returns

Grafo* Apontador para o novo grafo criado

```
00244 {
00245     bool inf;
00246     int distanciasFinais[MAX];
00247     int verticeAnt[MAX];
00248 
00249     int numVertices = ContadorVertices(g);
00250 
00251     Grafo* novo = CriarGrafo(&inf);
00252 
00253     //Cria os vertices
00254     for (int i = 1; i <= numVertices; i++)
00255     {
00256         novo = InserirVerticeGrafo(novo, i, &inf);
00257     }
00258 
00259     Vertice* aux = novo->inicioGrafo;
00260 
00261     //Quantos houver vertices
00262     while (aux)
00263     {
00264 
00265         InicializarArrays(distanciasFinais);
00266         Dijkstra(g, aux->id, distanciasFinais, verticeAnt); // Recebe um arrays com os valores mínimos
      de uma origem a todos os vertices
00267 
00268         for (int j = 1; j <= numVertices; j++)
00269         {
00270             //Coloca no grafo
00271             novo = InserirAdjGrafo(novo, aux->id, j, distanciasFinais[j], &inf);
00272         }
00273 
00274         aux = aux->nextV;
00275     }
00276 
00277     return novo;
00278 }
```

5.6.2.3 Dijkstra()

```
void Dijkstra (
            Grafo * g,
```

```
int origem,
int distanciasFinais[],
int verticeAnt[])
```

Algoritmo de Dijkstra.
Esta função implementa o algoritmo de Dijkstra, que é usado para encontrar os caminhos mais curtos de um vértice de origem para todos os outros vértices num grafo.

Parameters

g	Apontador para o grafo no qual o algoritmo de Dijkstra será executado
origem	O vértice de origem para o qual os caminhos mais curtos serão calculados
distanciasFinais	Array que será preenchido com as distâncias mais curtas da origem para cada vértice
verticeAnt	Array que será preenchido com os antecessores de cada vértice

```
00111 {
00112     if (g == NULL)
00113     {
00114         return;
00115     }
00116
00117     //Arrays temporario
00118     int distancias[MAX];
00119     bool inf, visitado[MAX] = { false };
00120
00121     InicializarArrays(distancias);
00122
00123     int numVertices = ContadorVertices(g);
00124
00125     //O peso das origens é sempre 0
00126     distancias[origem] = 0;
00127     distanciasFinais[origem] = 0;
00128
00129     for (int i = 0; i < numVertices; i++)
00130     {
00131         //Encontra a posição da adjacência com menor valor
00132         int verticeAtual = DistanciaMinima(distancias, visitado, numVertices);
00133
00134         visitado[verticeAtual] = true; //Coloca como visitado
00135
00136         //Colocamos no vertice
00137         Vertice* auxV = ColocaNumaPosicaoLista(g->inicioGrafo, verticeAtual, &inf);
00138         if (auxV == NULL) return;
00139         Adjacente* auxA = auxV->nextA;
00140
00141         //Avançamos com as adjacências
00142         while (auxA)
00143         {
00144             //Se o vertice não foi visitado e o peso anterior mais o atual é maior que o guardado
00145             if (!visitado[auxA->id] && distancias[verticeAtual] + auxA->peso < distancias[auxA->id])
00146             {
00147                 //Verifica se existe adjacências entre a origem e adjacência atual
00148                 if (ExisteAdjDoisVertices(auxV, verticeAtual, auxA->id))
00149                 {
00150                     //Soma e guarda o necessário
00151                     distancias[auxA->id] = distancias[verticeAtual] + auxA->peso;
00152                     distanciasFinais[auxA->id] = distancias[auxA->id];
00153                     verticeAnt[auxA->id] = verticeAtual;
00154                 }
00155             }
00156             auxA = auxA->next;
00157         }
00158     }
00159 }
```

5.6.2.4 DistanciaMinima()

```
int DistanciaMinima (
            int distancia[],
            bool visitado[],
            int n)
```

Encontra a o vértice com a distância mínima.
Esta função encontra o vértice com a distância mínima, a partir do conjunto de vértices ainda não incluídos no caminho mais curto. Retorna o id do vértice com a distância mínima.

Parameters

distancia	Array com as distâncias acumuladas de cada vértice.
visitado	Array que indica se um vértice foi ou não visitado.
n	Número total de vértices.

Returns

int O id do vértice com a distância mínima.

```
00076 {
00077     if (distancia == NULL || visitado == NULL)
00078     {
00079         return -1;
00080     }
00081
00082     int min = INT_MAX;
00083     int posicao = -1;
00084
00085     //Corre o arrays de booleanos verifica se o vértice já foi visitado e encontrar o valor mínimo das
      adjacências
00086     for (int i = 1; i <= n; i++)
00087     {
00088         if (visitado[i] == false && distancia[i] <= min)
00089         {
00090             min = distancia[i];
00091             posicao = i;
00092         }
00093     }
00094
00095     return posicao;
00096 }
```

5.6.2.5 DistanciaMinimaEntreVertices()

```
int DistanciaMinimaEntreVertices (
            Grafo * g,
            int origem,
            int destino)
```

Calcula a distância mínima entre dois vértices num grafo.
Esta função calcula a distância mínima entre dois vértices num grafo. A função retorna a distância mínima entre os vértices de origem e destino.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

int A distância mínima entre os vértices de origem e destino.

```
00173 {
00174     if (g == NULL)
00175     {
00176         return -1;
00177     }
00178
00179     int valor = 0;
00180     int distanciasFinais[MAX];
00181     int verticeAnt[MAX];
00182
00183     InicializarArrays(distanciasFinais);
00184     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00185
00186     //Verifica recebe o peso de uma adjacência
00187     if (distanciasFinais[destino] != INT_MAX && distanciasFinais[destino] > 0)
00188     {
00189         return distanciasFinais[destino];
00190     }
```

```
00191     else
00192     {
00193         return valor;
00194     }
00195
00196 }
```

5.6.2.6 ExisteCaminhoGrafo()

```
bool ExisteCaminhoGrafo (
            Grafo * g,
            int origem,
            int destino)
```

Verifica se existe um caminho entre dois vértices num grafo.
Esta função verifica se existe um caminho entre dois vértices num grafo. O grafo, o vértice de origem e o vértice de destino são passados como argumentos. A função retorna verdadeiro se existir um caminho válido entre os vértices de origem e destino, e falso caso contrário.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

true Se existir um caminho válido entre os vértices de origem e destino

false Se não existir um caminho válido entre os vértices de origem e destino

```
00212 {
00213     if (g == NULL)
00214     {
00215         return false;
00216     }
00217
00218     int verticeAnt[MAX];
00219     int distanciasFinais[MAX];
00220
00221     InicializarArrays(distanciasFinais);
00222     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00223
00224     //Verifica recebe o peso de uma adjacência e verifica se é valido
00225     if (distanciasFinais[destino] != INT_MAX && distanciasFinais[destino] > 0)
00226     {
00227         return true;
00228     }
00229     else
00230     {
00231         return false;
00232     }
00233 }
```

5.6.2.7 InicializarArrays()

```
void InicializarArrays (
            int * dis[])
```

Inicializa os arrays de distâncias.
Esta função inicializa os arrays de distâncias, atribuindo a cada elemento o valor máximo de um inteiro.

Parameters

dis	Array de distâncias a ser inicializado.

```
00022 {
00023     if (dis == NULL)
00024     {
00025         return;
00026     }
00027
```

```
00028     //Inicializa um arrays no valor maximo de um inteiro
00029     for (int i = 0; i < MAX; i++)
00030     {
00031         dis[i] = INT_MAX;
00032     }
00033 }
```

5.7 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/caminhos.h File Reference

Este arquivo de cabeçalho define as estruturas de dados e as funções para manipular caminhos num grafo.
`#include "grafo.h"`

Functions

- void InicializarArrays (int *dis[ ])
  Inicializa os arrays de distâncias.
- int ContadorVertices (Grafo *g)
  Conta o número de vértices num grafo.
- int DistanciaMinima (int distancia[ ], bool visitado[ ], int n)
  Encontra a o vértice com a distância mínima.
- void Dijkstra (Grafo *g, int origem, int distanciasFinais[ ])
  Algoritmo de Dijkstra.
- int DistanciaMinimaEntreVertices (Grafo *g, int origem, int destino)
  Calcula a distância mínima entre dois vértices num grafo.
- bool ExisteCaminhoGrafo (Grafo *g, int origem, int destino)
  Verifica se existe um caminho entre dois vértices num grafo.
- Grafo * CriarGrafoCaminhoMaisCurto (Grafo *g)
  Cria um grafo com os caminhos mais curto para cada vertice.

5.7.1 Detailed Description

Este arquivo de cabeçalho define as estruturas de dados e as funções para manipular caminhos num grafo.

Author

Hugo Cruz (a23010)

Version

48.1

Date

2024-05-24

Copyright

5.7.2 Function Documentation

5.7.2.1 ContadorVertices()

```
int ContadorVertices (
            Grafo * g)
```

Conta o número de vértices num grafo.
Esta função conta o número de vértices num grafo. Retorna o número de vértices no grafo.

Parameters

g	O grafo a ser analisado.

Returns

int O número de vértices no grafo.

```
00044 {
00045     if (g == NULL)
00046     {
00047         return 0;
00048     }
00049
00050     int contador = 0;
00051
00052     Vertice* aux = g->inicioGrafo;
00053
00054     //Avança na lista e conta o número de vertices.
00055     while (aux)
00056     {
00057         contador++;
00058         aux = aux->nextV;
00059     }
00060
00061     return contador;
00062 }
```

5.7.2.2 CriarGrafoCaminhoMaisCurto()

```
Grafo * CriarGrafoCaminhoMaisCurto (
            Grafo * g)
```

Cria um grafo com os caminhos mais curto para cada vertice.

Parameters

g	Apontador para o grafo original

Returns

Grafo* Apontador para o novo grafo criado

Esta função cria um novo grafo com os caminhos mais curtos para cada vértice

Parameters

g	Apontador para o grafo original

Returns

Grafo* Apontador para o novo grafo criado

```
00244 {
00245     bool inf;
00246     int distanciasFinais[MAX];
00247     int verticeAnt[MAX];
00248
00249     int numVertices = ContadorVertices(g);
00250
00251     Grafo* novo = CriarGrafo(&inf);
00252
00253     //Cria os vertices
00254     for (int i = 1; i <= numVertices; i++)
00255     {
00256         novo = InserirVerticeGrafo(novo, i, &inf);
00257     }
00258
00259     Vertice* aux = novo->inicioGrafo;
00260
00261     //Quantos houver vertices
00262     while (aux)
00263     {
```

```
00264
00265         InicializarArrays(distanciasFinais);
00266         Dijkstra(g, aux->id, distanciasFinais, verticeAnt); // Recebe um arrays com os valores mínimos
      de uma origem a todos os vertices
00267
00268         for (int j = 1; j <= numVertices; j++)
00269         {
00270             //Coloca no grafo
00271             novo = InserirAdjGrafo(novo, aux->id, j, distanciasFinais[j], &inf);
00272         }
00273
00274         aux = aux->nextV;
00275     }
00276
00277     return novo;
00278 }
```

5.7.2.3 Dijkstra()

```
void Dijkstra (
            Grafo * g,
            int origem,
            int distanciasFinais[])
```

Algoritmo de Dijkstra.
Esta função implementa o algoritmo de Dijkstra, que é usado para encontrar os caminhos mais curtos de um vértice de origem para todos os outros vértices num grafo.

Parameters

g	Apontador para o grafo no qual o algoritmo de Dijkstra será executado
origem	O vértice de origem para o qual os caminhos mais curtos serão calculados
distanciasFinais	Array que será preenchido com as distâncias mais curtas da origem para cada vértice

5.7.2.4 DistanciaMinima()

```
int DistanciaMinima (
            int distancia[],
            bool visitado[],
            int n)
```

Encontra a o vértice com a distância mínima.
Esta função encontra o vértice com a distância mínima, a partir do conjunto de vértices ainda não incluídos no caminho mais curto. Retorna o id do vértice com a distância mínima.

Parameters

distancia	Array com as distâncias acumuladas de cada vértice.
visitado	Array que indica se um vértice foi ou não visitado.
n	Número total de vértices.

Returns

int O id do vértice com a distância mínima.

```
00076 {
00077     if (distancia == NULL || visitado == NULL)
00078     {
00079         return -1;
00080     }
00081
00082     int min = INT_MAX;
00083     int posicao = -1;
00084
00085     //Corre o arrays de booleanos verifica se o vértice já foi visitado e encontrar o valor mínimo das
      adjacências
00086     for (int i = 1; i <= n; i++)
00087     {
```

```
00088         if (visitado[i] == false && distancia[i] <= min)
00089         {
00090             min = distancia[i];
00091             posicao = i;
00092         }
00093     }
00094
00095     return posicao;
00096 }
```

5.7.2.5 DistanciaMinimaEntreVertices()

```
int DistanciaMinimaEntreVertices (
            Grafo * g,
            int origem,
            int destino)
```

Calcula a distância mínima entre dois vértices num grafo.
Esta função calcula a distância mínima entre dois vértices num grafo. O grafo, o vértice de origem e o vértice de destino são passados como argumentos. A função retorna a distância mínima entre os vértices de origem e destino.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

int A distância mínima entre os vértices de origem e destino.

Esta função calcula a distância mínima entre dois vértices num grafo. A função retorna a distância mínima entre os vértices de origem e destino.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

int A distância mínima entre os vértices de origem e destino.

```
00173 {
00174     if (g == NULL)
00175     {
00176         return -1;
00177     }
00178
00179     int valor = 0;
00180     int distanciasFinais[MAX];
00181     int verticeAnt[MAX];
00182
00183     InicializarArrays(distanciasFinais);
00184     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00185
00186     //Verifica recebe o peso de uma adjacência
00187     if (distanciasFinais[destino] != INT_MAX && distanciasFinais[destino] > 0)
00188     {
00189         return distanciasFinais[destino];
00190     }
00191     else
00192     {
00193         return valor;
00194     }
00195
00196 }
```

5.7.2.6 ExisteCaminhoGrafo()

```
bool ExisteCaminhoGrafo (
            Grafo * g,
            int origem,
            int destino)
```

Verifica se existe um caminho entre dois vértices num grafo.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

true Se existir um caminho válido entre os vértices de origem e destino

false Se não existir um caminho válido entre os vértices de origem e destino

Esta função verifica se existe um caminho entre dois vértices num grafo. O grafo, o vértice de origem e o vértice de destino são passados como argumentos. A função retorna verdadeiro se existir um caminho válido entre os vértices de origem e destino, e falso caso contrário.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

true Se existir um caminho válido entre os vértices de origem e destino

false Se não existir um caminho válido entre os vértices de origem e destino

```
00212 {
00213     if (g == NULL)
00214     {
00215         return false;
00216     }
00217
00218     int verticeAnt[MAX];
00219     int distanciasFinais[MAX];
00220
00221     InicializarArrays(distanciasFinais);
00222     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00223
00224     //Verifica recebe o peso de uma adjacência e verifica se é valido
00225     if (distanciasFinais[destino] != INT_MAX && distanciasFinais[destino] > 0)
00226     {
00227         return true;
00228     }
00229     else
00230     {
00231         return false;
00232     }
00233 }
```

5.7.2.7 InicializarArrays()

```
void InicializarArrays (
            int * dis[])
```

Inicializa os arrays de distâncias.
Esta função inicializa os arrays de distâncias, atribuindo a cada elemento o valor máximo de um inteiro.

Parameters

dis	Array de distâncias a ser inicializado.

```
00022 {
00023     if (dis == NULL)
00024     {
00025         return;
00026     }
00027
00028     //Inicializa um arrays no valor maximo de um inteiro
00029     for (int i = 0; i < MAX; i++)
00030     {
00031         dis[i] = INT_MAX;
00032     }
00033 }
```

5.8 caminhos.h

Go to the documentation of this file.

```
00001
00011 #ifndef CAMINHOS_H
00012 #define CAMINHOS_H
00013
00014 #include "grafo.h"
00015
00023 void InicializarArrays(int* dis[]);
00024
00033 int ContadorVertices(Grafo* g);
00034
00046 int DistanciaMinima(int distancia[], bool visitado[], int n);
00047
00057 void Dijkstra(Grafo* g, int origem, int distanciasFinais[]);
00058
00070 int DistanciaMinimaEntreVertices(Grafo* g, int origem, int destino);
00071
00081 bool ExisteCaminhoGrafo(Grafo* g, int origem, int destino);
00082
00089 Grafo* CriarGrafoCaminhoMaisCurto(Grafo* g);
00090
00091 #endif
```

5.9 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Main/libs/caminhos.h File Reference

Este arquivo de cabeçalho define as estruturas de dados e as funções para manipular caminhos num grafo.

`#include "grafo.h"`

Functions

- void InicializarArrays (int ∗dis[ ])

  Inicializa os arrays de distâncias.
- int ContadorVertices (Grafo ∗g)

  Conta o número de vértices num grafo.
- int DistanciaMinima (int distancia[ ], bool visitado[ ], int n)

  Encontra a o vértice com a distância mínima.
- void Dijkstra (Grafo ∗g, int origem, int distanciasFinais[ ])

  Algoritmo de Dijkstra.
- int DistanciaMinimaEntreVertices (Grafo ∗g, int origem, int destino)

  Calcula a distância mínima entre dois vértices num grafo.
- bool ExisteCaminhoGrafo (Grafo ∗g, int origem, int destino)

  Verifica se existe um caminho entre dois vértices num grafo.
- Grafo ∗ CriarGrafoCaminhoMaisCurto (Grafo ∗g)

  Cria um grafo com os caminhos mais curto para cada vertice.

5.9.1 Detailed Description

Este arquivo de cabeçalho define as estruturas de dados e as funções para manipular caminhos num grafo.

Author

Hugo Cruz (a23010)

Version

48.1

Date

2024-05-24

Copyright

5.9.2 Function Documentation

5.9.2.1 ContadorVertices()

```
int ContadorVertices (
            Grafo * g)
```

Conta o número de vértices num grafo.
Esta função conta o número de vértices num grafo. Retorna o número de vértices no grafo.

Parameters

g	O grafo a ser analisado.

Returns

int O número de vértices no grafo.

```
00044 {
00045     if (g == NULL)
00046     {
00047         return 0;
00048     }
00049
00050     int contador = 0;
00051
00052     Vertice* aux = g->inicioGrafo;
00053
00054     //Avança na lista e conta o número de vertices.
00055     while (aux)
00056     {
00057         contador++;
00058         aux = aux->nextV;
00059     }
00060
00061     return contador;
00062 }
```

5.9.2.2 CriarGrafoCaminhoMaisCurto()

```
Grafo * CriarGrafoCaminhoMaisCurto (
            Grafo * g)
```

Cria um grafo com os caminhos mais curto para cada vertice.

Parameters

g	Apontador para o grafo original

Returns

Grafo∗ Apontador para o novo grafo criado

Esta função cria um novo grafo com os caminhos mais curtos para cada vértice

Parameters

g	Apontador para o grafo original

Returns

Grafo* Apontador para o novo grafo criado

```
00244 {
00245     bool inf;
00246     int distanciasFinais[MAX];
00247     int verticeAnt[MAX];
00248 
00249     int numVertices = ContadorVertices(g);
00250 
00251     Grafo* novo = CriarGrafo(&inf);
00252 
00253     //Cria os vertices
00254     for (int i = 1; i <= numVertices; i++)
00255     {
00256         novo = InserirVerticeGrafo(novo, i, &inf);
00257     }
00258 
00259     Vertice* aux = novo->inicioGrafo;
00260 
00261     //Quantos houver vertices
00262     while (aux)
00263     {
00264 
00265         InicializarArrays(distanciasFinais);
00266         Dijkstra(g, aux->id, distanciasFinais, verticeAnt); // Recebe um arrays com os valores mínimos
      de uma origem a todos os vertices
00267 
00268         for (int j = 1; j <= numVertices; j++)
00269         {
00270             //Coloca no grafo
00271             novo = InserirAdjGrafo(novo, aux->id, j, distanciasFinais[j], &inf);
00272         }
00273 
00274         aux = aux->nextV;
00275     }
00276 
00277     return novo;
00278 }
```

5.9.2.3 Dijkstra()

```
void Dijkstra (
            Grafo * g,
            int origem,
            int distanciasFinais[])
```

Algoritmo de Dijkstra.
Esta função implementa o algoritmo de Dijkstra, que é usado para encontrar os caminhos mais curtos de um vértice de origem para todos os outros vértices num grafo.

Parameters

g	Apontador para o grafo no qual o algoritmo de Dijkstra será executado
origem	O vértice de origem para o qual os caminhos mais curtos serão calculados
distanciasFinais	Array que será preenchido com as distâncias mais curtas da origem para cada vértice

5.9.2.4 DistanciaMinima()

```
int DistanciaMinima (
            int distancia[],
            bool visitado[],
            int n)
```

Encontra a o vértice com a distância mínima.
Esta função encontra o vértice com a distância mínima, a partir do conjunto de vértices ainda não incluídos no caminho mais curto. Retorna o id do vértice com a distância mínima.

Parameters

distancia	Array com as distâncias acumuladas de cada vértice.
visitado	Array que indica se um vértice foi ou não visitado.
n	Número total de vértices.

Returns

int O id do vértice com a distância mínima.

```
00076 {
00077     if (distancia == NULL || visitado == NULL)
00078     {
00079         return -1;
00080     }
00081
00082     int min = INT_MAX;
00083     int posicao = -1;
00084
00085     //Corre o arrays de booleanos verifica se o vértice já foi visitado e encontrar o valor mínimo das
      adjacências
00086     for (int i = 1; i <= n; i++)
00087     {
00088         if (visitado[i] == false && distancia[i] <= min)
00089         {
00090             min = distancia[i];
00091             posicao = i;
00092         }
00093     }
00094
00095     return posicao;
00096 }
```

5.9.2.5 DistanciaMinimaEntreVertices()

```
int DistanciaMinimaEntreVertices (
            Grafo * g,
            int origem,
            int destino)
```

Calcula a distância mínima entre dois vértices num grafo.
Esta função calcula a distância mínima entre dois vértices num grafo. O grafo, o vértice de origem e o vértice de destino são passados como argumentos. A função retorna a distância mínima entre os vértices de origem e destino.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

int A distância mínima entre os vértices de origem e destino.

Esta função calcula a distância mínima entre dois vértices num grafo. A função retorna a distância mínima entre os vértices de origem e destino.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

int A distância mínima entre os vértices de origem e destino.

```
00173 {
00174     if (g == NULL)
00175     {
00176         return -1;
00177     }
00178 
00179     int valor = 0;
00180     int distanciasFinais[MAX];
00181     int verticeAnt[MAX];
00182 
00183     InicializarArrays(distanciasFinais);
00184     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00185 
00186     //Verifica recebe o peso de uma adjacência
00187     if (distanciasFinais[destino] != INT_MAX && distanciasFinais[destino] > 0)
00188     {
00189         return distanciasFinais[destino];
00190     }
00191     else
00192     {
00193         return valor;
00194     }
00195 
00196 }
```

5.9.2.6 ExisteCaminhoGrafo()

```
bool ExisteCaminhoGrafo (
            Grafo * g,
            int origem,
            int destino)
```

Verifica se existe um caminho entre dois vértices num grafo.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

true Se existir um caminho válido entre os vértices de origem e destino

false Se não existir um caminho válido entre os vértices de origem e destino

Esta função verifica se existe um caminho entre dois vértices num grafo. O grafo, o vértice de origem e o vértice de destino são passados como argumentos. A função retorna verdadeiro se existir um caminho válido entre os vértices de origem e destino, e falso caso contrário.

Parameters

g	Apontador para o grafo onde a operação será realizada
origem	Vértice de origem
destino	Vértice de destino

Returns

true Se existir um caminho válido entre os vértices de origem e destino

false Se não existir um caminho válido entre os vértices de origem e destino

```
00212 {
00213     if (g == NULL)
00214     {
00215         return false;
00216     }
```

```
00217
00218     int verticeAnt[MAX];
00219     int distanciasFinais[MAX];
00220
00221     InicializarArrays(distanciasFinais);
00222     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00223
00224     //Verifica recebe o peso de uma adjacência e verifica se é valido
00225     if (distanciasFinais[destino] != INT_MAX && distanciasFinais[destino] > 0)
00226     {
00227         return true;
00228     }
00229     else
00230     {
00231         return false;
00232     }
00233 }
```

5.9.2.7 InicializarArrays()

```
void InicializarArrays (
             int * dis[])
```

Inicializa os arrays de distâncias.
Esta função inicializa os arrays de distâncias, atribuindo a cada elemento o valor máximo de um inteiro.

Parameters

dis	Array de distâncias a ser inicializado.

```
00022 {
00023     if (dis == NULL)
00024     {
00025         return;
00026     }
00027
00028     //Inicializa um arrays no valor maximo de um inteiro
00029     for (int i = 0; i < MAX; i++)
00030     {
00031         dis[i] = INT_MAX;
00032     }
00033 }
```

5.10 caminhos.h

Go to the documentation of this file.

```
00001
00011 #ifndef CAMINHOS_H
00012 #define CAMINHOS_H
00013
00014 #include "grafo.h"
00015
00023 void InicializarArrays(int* dis[]);
00024
00033 int ContadorVertices(Grafo* g);
00034
00046 int DistanciaMinima(int distancia[], bool visitado[], int n);
00047
00057 void Dijkstra(Grafo* g, int origem, int distanciasFinais[]);
00058
00070 int DistanciaMinimaEntreVertices(Grafo* g, int origem, int destino);
00071
00081 bool ExisteCaminhoGrafo(Grafo* g, int origem, int destino);
00082
00089 Grafo* CriarGrafoCaminhoMaisCurto(Grafo* g);
00090
00091 #endif
```

5.11 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/grafo.c File Reference

Ficheiro de implementação das funções que manipulam a estrutura de dados Grafo.

```
#include "grafo.h"
```

Functions

- void ApagaGrafo (Grafo *g)

  Apaga um grafo e limpa a memória alocada.
- Grafo * CriarGrafo (bool *inf)

  Cria um novo grafo.
- Grafo * InserirVerticeGrafo (Grafo *g, int novo, bool *inf)

  Insere um novo vértice no grafo.
- Grafo * InserirAdjGrafo (Grafo *g, int origem, int destino, int peso, bool *inf)

  Insere uma nova aresta no grafo.
- Grafo * EliminaVerticeGrafo (Grafo *g, int id, bool *inf)

  Elimina um vértice do grafo.
- Grafo * EliminaAdjGrafo (Grafo *g, int origem, int destino, bool *inf)

  Elimina uma aresta do grafo.
- bool ExisteAdjDoisVertices (Vertice *inicio, int origem, int destino)

  Verifica se existe uma aresta entre dois vértices.

5.11.1 Detailed Description

Ficheiro de implementação das funções que manipulam a estrutura de dados Grafo.

Author

Hugo Cruz (a23010)

Version

156.1

Date

2024-05-24

Copyright

Copyright (c) 2024

5.11.2 Function Documentation

5.11.2.1 ApagaGrafo()

```
void ApagaGrafo (
            Grafo * g)
```

Apaga um grafo e limpa a memória alocada.
Esta função apaga um grafo e liberta a memória alocada. Ela percorre todos os vértices e adjacências do grafo, apagando-os um a um

Parameters

g	Apontador para o grafo a ser apagado

```
00022 {
00023     if (g == NULL)
00024         return;
00025
00026     Vertice* v = g->inicioGrafo;
00027
```

```
00028     //Apagar todos os vertices e adjacências
00029     while (v != NULL)
00030     {
00031         Adjacente* a = v->nextA;
00032         while (a != NULL)
00033         {
00034             Adjacente* temp = a;
00035             a = a->next;
00036             ApagarAdjacencia(temp);
00037         }
00038 
00039         Vertice* tempV = v;
00040         v = v->nextV;
00041         ApagarVertice(tempV);
00042     }
00043 
00044     free(g);
00045 }
```

5.11.2.2 CriarGrafo()

```
Grafo * CriarGrafo (
            bool * inf)
```

Cria um novo grafo.

Esta função cria um novo grafo. Ela aloca memória para um novo grafo e retorna um apontador para o grafo criado.

Parameters

inf	Apontador para uma variável booleana que será definida como true se a criação do grafo for bem-sucedida, e false caso contrário.

Returns

Apontador para o novo grafo criado.

```
00057 {
00058     *inf = false;
00059 
00060     Grafo* aux = (Grafo*)malloc(sizeof(Grafo)); //Aloca memória para um grafo
00061 
00062     if (aux == NULL)
00063     {
00064         *inf = false;
00065         return NULL;
00066     }
00067 
00068     aux->inicioGrafo = NULL;
00069 
00070     *inf = true;
00071     return aux;
00072 }
```

5.11.2.3 EliminaAdjGrafo()

```
Grafo * EliminaAdjGrafo (
            Grafo * g,
            int origem,
            int destino,
            bool * inf)
```

Elimina uma aresta do grafo.

Esta função elimina uma aresta do grafo. Ela verifica se a aresta a ser eliminada existe e, em seguida, remove-a do grafo.

Parameters

g	Apontador para o grafo do qual a aresta será eliminada.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.
inf	Apontador para uma variável booleana que será definida como true se a eliminação for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com a aresta eliminada.

```
00194 {
00195     *inf = false;
00196
00197     if (g == NULL)
00198     {
00199         return NULL;
00200     }
00201
00202     //Coloca se no posição de origem
00203     Vertice* origemVertice = ColocaNumaPosicaoLista(g->inicioGrafo, origem, &inf);
00204
00205     if (origemVertice == NULL)
00206     {
00207         *inf = false;
00208         return g;
00209     }
00210
00211     //Coloca se no posição de destino
00212     Vertice* destinoVertice = ColocaNumaPosicaoLista(g->inicioGrafo, destino, &inf);
00213
00214     if (destinoVertice == NULL)
00215     {
00216         *inf = false;
00217         return g;
00218     }
00219
00220     //Apaga um adjacência entre dois vértices
00221     origemVertice->nextA = EliminaUmaAdj(origemVertice->nextA, destino, &inf);
00222
00223     return g;
00224 }
```

5.11.2.4 EliminaVerticeGrafo()

```
Grafo * EliminaVerticeGrafo (
            Grafo * g,
            int id,
            bool * inf)
```

Elimina um vértice do grafo.
Esta função elimina um vértice do grafo. Ela verifica se o vértice a ser eliminado existe e, em seguida, remove-o do grafo.

Parameters

g	Apontador para o grafo do qual o vértice será eliminado.
id	O valor do vértice a ser eliminado.
inf	Apontador para uma variável booleana que será definida como true se a eliminação for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com o vértice eliminado.

```
00165 {
00166     *inf = false;
00167
00168     if (g == NULL )
00169     {
00170         return NULL;
00171     }
00172
00173     //Elimina o vértice da lista de vertices do grafo
00174     g->inicioGrafo = EliminarVertice(g->inicioGrafo, id, &inf);
00175     //Apaga todas as adjacências relacionadas com o id
00176     g->inicioGrafo = EliminarTodasAdjacenciasVertice(g->inicioGrafo, id, &inf);
00177
00178     return g;
00179 }
```

5.11.2.5 ExisteAdjDoisVertices()

```
bool ExisteAdjDoisVertices (
            Vertice * inicio,
            int origem,
            int destino)
```

Verifica se existe uma aresta entre dois vértices.
Esta função verifica se existe uma aresta entre dois vértices. Ela percorre todos os vértices e adjacências do grafo, procurando uma aresta entre os vértices de origem e destino.

Parameters

inicio	Apontador para o primeiro vértice do grafo.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.

Returns

bool Retorna true se existe uma aresta entre os vértices origem e destino, e false caso contrário.

```
00238 {
00239     if (inicio == NULL )
00240     {
00241         return false;
00242     }
00243
00244     if (ExisteVertice(inicio, origem))
00245     {
00246         Vertice* auxV = inicio;
00247
00248         while (auxV)
00249         {
00250             Adjacente* auxA = auxV->nextA;
00251
00252             while (auxA)
00253             {
00254                 //Procura dois vértices e  informa se os mesmos existem
00255                 if (auxV->id == origem && auxA->id == destino)
00256                 {
00257                     if (auxA->peso > 0)
00258                     {
00259                         return true;
00260                     }
00261                 }
00262
00263                 auxA = auxA->next;
00264             }
00265
00266             auxV = auxV->nextV;
00267         }
00268
00269         return false;
00270     }
00271     else
00272     {
00273         return false;
00274     }
00275 }
```

5.11.2.6 InserirAdjGrafo()

```
Grafo * InserirAdjGrafo (
            Grafo * g,
            int origem,
            int destino,
            int peso,
            bool * inf)
```

Insere uma nova aresta no grafo.
Esta função insere uma nova aresta no grafo. Ela verifica se os vértices de origem e destino são válidos e, em seguida, insere a aresta entre eles.

Parameters

g	Apontador para o grafo onde a aresta será inserida.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.
peso	O peso da aresta a ser inserida.
inf	Apontador para uma variável booleana que será definida como true se a inserção for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com a nova aresta inserida.

```
00121 {
00122     *inf = false;
00123
00124     if (g == NULL)
00125     {
00126         return NULL;
00127     }
00128
00129     //Coloca se no posição de origem
00130     Vertice* origemVertice = ColocaNumaPosicaoLista(g->inicioGrafo, origem, &inf);
00131
00132     if (origemVertice == NULL)
00133     {
00134         *inf = false;
00135         return g;
00136     }
00137
00138     //Coloca se no posição de destino
00139     Vertice* destinoVertice = ColocaNumaPosicaoLista(g->inicioGrafo, destino, &inf);
00140
00141     if (destinoVertice == NULL)
00142     {
00143         *inf = false;
00144         return g;
00145     }
00146
00147     //Insere uma adjacência
00148     origemVertice->nextA = InserirAdjacenciaLista(origemVertice->nextA, destino, peso, &inf);
00149
00150     return g;
00151 }
```

5.11.2.7 InserirVerticeGrafo()

```
Grafo * InserirVerticeGrafo (
            Grafo * g,
            int novo,
            bool * inf)
```

Insere um novo vértice no grafo.
Esta função insere um novo vértice no grafo. Ela verifica se o valor do novo vértice já existe e, em seguida, insere-o no fim da lista de vertices.

Parameters

g	Apontador para o grafo onde o vértice será inserido.
novo	O valor do novo vértice a ser inserido.
inf	Apontador para uma variável booleana que será definida como true se a inserção for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com o novo vértice inserido.

```
00086 {
00087     *inf = false;
00088
```

```
00089     if (g == NULL)
00090     {
00091         return NULL;
00092     }
00093     else
00094     {
00095         //Insere o novo vértice na lista de vertices do grafo
00096         g->inicioGrafo = InserirVerticeLista(g->inicioGrafo, novo, &inf);
00097 
00098         if (g->inicioGrafo != NULL)
00099         {
00100             *inf = true;
00101         }
00102     }
00103 
00104     return g;
00105 }
```

5.12 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/grafo.h File Reference

Ficheiro de cabeçalho para a estrutura de dados Grafo e funcionalidades.

```
#include "vertices.h"
```

Data Structures

- struct Grafo

  Estrutura de dados para um Grafo.

Macros

- #define MAX 10

  Define um valor.

Typedefs

- typedef struct Grafo Grafo

  Estrutura de dados para um Grafo.

Functions

- void ApagaGrafo (Grafo *g)

  Apaga um grafo e limpa a memória alocada.
- Grafo * CriarGrafo (bool *inf)

  Cria um novo grafo.
- Grafo * InserirVerticeGrafo (Grafo *g, int novo, bool *inf)

  Insere um novo vértice no grafo.
- Grafo * InserirAdjGrafo (Grafo *g, int origem, int destino, int peso, bool *inf)

  Insere uma nova aresta no grafo.
- Grafo * EliminaVerticeGrafo (Grafo *g, int id, bool *inf)

  Elimina um vértice do grafo.
- Grafo * EliminaAdjGrafo (Grafo *g, int origem, int destino, bool *inf)

  Elimina uma aresta do grafo.
- bool ExisteAdjDoisVertices (Vertice *inicio, int origem, int destino)

  Verifica se existe uma aresta entre dois vértices.

5.12.1 Detailed Description

Ficheiro de cabeçalho para a estrutura de dados Grafo e funcionalidades.

Author

Hugo Cruz (a23010)

Version

88.1

Date

2024-05-24

Copyright

Copyright (c) 2024

5.12.2 Macro Definition Documentation

5.12.2.1 MAX

```
#define MAX 10
```

Define um valor.
Esta macro define um valor maximo de 10.

5.12.3 Typedef Documentation

5.12.3.1 Grafo

```
typedef struct Grafo Grafo
```

Estrutura de dados para um Grafo.
A estrutura Grafo é uma representação de um grafo em memória, onde inicioGrafo é um apontador para o primeiro vértice do grafo.

5.12.4 Function Documentation

5.12.4.1 ApagaGrafo()

```
void ApagaGrafo (
            Grafo * g)
```

Apaga um grafo e limpa a memória alocada.
Esta função apaga um grafo e liberta a memória alocada. Ela percorre todos os vértices e adjacências do grafo, apagando-os um a um

Parameters

g	Apontador para o grafo a ser apagado

```
00022 {
00023     if (g == NULL)
00024         return;
00025
00026     Vertice* v = g->inicioGrafo;
00027
00028     //Apagar todos os vertices e adjacências
00029     while (v != NULL)
00030     {
00031         Adjacente* a = v->nextA;
00032         while (a != NULL)
00033         {
00034             Adjacente* temp = a;
00035             a = a->next;
00036             ApagarAdjacencia(temp);
00037         }
```

```
00038
00039         Vertice* tempV = v;
00040         v = v->nextV;
00041         ApagarVertice(tempV);
00042     }
00043
00044     free(g);
00045 }
```

5.12.4.2 CriarGrafo()

```
Grafo * CriarGrafo (
            bool * inf)
```

Cria um novo grafo.
Esta função cria um novo grafo. Ela aloca memória para um novo grafo e retorna um apontador para o grafo criado.

Parameters

inf	Apontador para uma variável booleana que será definida como true se a criação do grafo for bem-sucedida, e false caso contrário.

Returns

Apontador para o novo grafo criado.

```
00057 {
00058     *inf = false;
00059
00060     Grafo* aux = (Grafo*)malloc(sizeof(Grafo)); //Aloca memória para um grafo
00061
00062     if (aux == NULL)
00063     {
00064         *inf = false;
00065         return NULL;
00066     }
00067
00068     aux->inicioGrafo = NULL;
00069
00070     *inf = true;
00071     return aux;
00072 }
```

5.12.4.3 EliminaAdjGrafo()

```
Grafo * EliminaAdjGrafo (
            Grafo * g,
            int origem,
            int destino,
            bool * inf)
```

Elimina uma aresta do grafo.
Esta função elimina uma aresta do grafo. Ela verifica se a aresta a ser eliminada existe e, em seguida, remove-a do grafo.

Parameters

g	Apontador para o grafo do qual a aresta será eliminada.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.
inf	Apontador para uma variável booleana que será definida como true se a eliminação for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com a aresta eliminada.

```
00194 {
00195     *inf = false;
00196
00197     if (g == NULL)
00198     {
00199         return NULL;
00200     }
00201
00202     //Coloca se no posição de origem
00203     Vertice* origemVertice = ColocaNumaPosicaoLista(g->inicioGrafo, origem, &inf);
00204
00205     if (origemVertice == NULL)
00206     {
00207         *inf = false;
00208         return g;
00209     }
00210
00211     //Coloca se no posição de destino
00212     Vertice* destinoVertice = ColocaNumaPosicaoLista(g->inicioGrafo, destino, &inf);
00213
00214     if (destinoVertice == NULL)
00215     {
00216         *inf = false;
00217         return g;
00218     }
00219
00220     //Apaga um adjacência entre dois vértices
00221     origemVertice->nextA = EliminaUmaAdj(origemVertice->nextA, destino, &inf);
00222
00223     return g;
00224 }
```

5.12.4.4 EliminaVerticeGrafo()

```
Grafo * EliminaVerticeGrafo (
            Grafo * g,
            int id,
            bool * inf)
```

Elimina um vértice do grafo.
Esta função elimina um vértice do grafo. Ela verifica se o vértice a ser eliminado existe e, em seguida, remove-o do grafo.

Parameters

g	Apontador para o grafo do qual o vértice será eliminado.
id	O valor do vértice a ser eliminado.
inf	Apontador para uma variável booleana que será definida como true se a eliminação for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com o vértice eliminado.

```
00165 {
00166     *inf = false;
00167
00168     if (g == NULL )
00169     {
00170         return NULL;
00171     }
00172
00173     //Elimina o vértice da lista de vertices do grafo
00174     g->inicioGrafo = EliminarVertice(g->inicioGrafo, id, &inf);
00175     //Apaga todas as adjacências relacionadas com o id
00176     g->inicioGrafo = EliminarTodasAdjacenciasVertice(g->inicioGrafo, id, &inf);
00177
00178     return g;
00179 }
```

5.12.4.5 ExisteAdjDoisVertices()

```
bool ExisteAdjDoisVertices (
            Vertice * inicio,
            int origem,
            int destino)
```

Verifica se existe uma aresta entre dois vértices.
Esta função verifica se existe uma aresta entre dois vértices. Ela percorre todos os vértices e adjacências do grafo, procurando uma aresta entre os vértices de origem e destino.

Parameters

inicio	Apontador para o primeiro vértice do grafo.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.

Returns

bool Retorna true se existe uma aresta entre os vértices origem e destino, e false caso contrário.

```
00238 {
00239     if (inicio == NULL )
00240     {
00241         return false;
00242     }
00243
00244     if (ExisteVertice(inicio, origem))
00245     {
00246         Vertice* auxV = inicio;
00247
00248         while (auxV)
00249         {
00250             Adjacente* auxA = auxV->nextA;
00251
00252             while (auxA)
00253             {
00254                 //Procura dois vértices e  informa se os mesmos existem
00255                 if (auxV->id == origem && auxA->id == destino)
00256                 {
00257                     if (auxA->peso > 0)
00258                     {
00259                         return true;
00260                     }
00261                 }
00262
00263                 auxA = auxA->next;
00264             }
00265
00266             auxV = auxV->nextV;
00267         }
00268
00269         return false;
00270     }
00271     else
00272     {
00273         return false;
00274     }
00275 }
```

5.12.4.6 InserirAdjGrafo()

```
Grafo * InserirAdjGrafo (
            Grafo * g,
            int origem,
            int destino,
            int peso,
            bool * inf)
```

Insere uma nova aresta no grafo.
Esta função insere uma nova aresta no grafo. Ela verifica se os vértices de origem e destino são válidos e, em seguida, insere a aresta entre eles.

Parameters

g	Apontador para o grafo onde a aresta será inserida.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.
peso	O peso da aresta a ser inserida.
inf	Apontador para uma variável booleana que será definida como true se a inserção for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com a nova aresta inserida.

```
00121 {
00122     *inf = false;
00123
00124     if (g == NULL)
00125     {
00126         return NULL;
00127     }
00128
00129     //Coloca se no posição de origem
00130     Vertice* origemVertice = ColocaNumaPosicaoLista(g->inicioGrafo, origem, &inf);
00131
00132     if (origemVertice == NULL)
00133     {
00134         *inf = false;
00135         return g;
00136     }
00137
00138     //Coloca se no posição de destino
00139     Vertice* destinoVertice = ColocaNumaPosicaoLista(g->inicioGrafo, destino, &inf);
00140
00141     if (destinoVertice == NULL)
00142     {
00143         *inf = false;
00144         return g;
00145     }
00146
00147     //Insere uma adjacência
00148     origemVertice->nextA = InserirAdjacenciaLista(origemVertice->nextA, destino, peso, &inf);
00149
00150     return g;
00151 }
```

5.12.4.7 InserirVerticeGrafo()

```
Grafo * InserirVerticeGrafo (
            Grafo * g,
            int novo,
            bool * inf)
```

Insere um novo vértice no grafo.
Esta função insere um novo vértice no grafo. Ela verifica se o valor do novo vértice já existe e, em seguida, insere-o no fim da lista de vertices.

Parameters

g	Apontador para o grafo onde o vértice será inserido.
novo	O valor do novo vértice a ser inserido.
inf	Apontador para uma variável booleana que será definida como true se a inserção for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com o novo vértice inserido.

```
00086 {
00087     *inf = false;
00088
```

```
00089     if (g == NULL)
00090     {
00091         return NULL;
00092     }
00093     else
00094     {
00095         //Insere o novo vértice na lista de vertices do grafo
00096         g->inicioGrafo = InserirVerticeLista(g->inicioGrafo, novo, &inf);
00097 
00098         if (g->inicioGrafo != NULL)
00099         {
00100             *inf = true;
00101         }
00102     }
00103 
00104     return g;
00105 }
```

5.13 grafo.h

Go to the documentation of this file.

```
00001 
00012 #ifndef GRAFO_H
00013 #define GRAFO_H
00014 
00021 #define MAX 10
00022 
00023 #include "vertices.h"
00024 
00033 typedef struct Grafo
00034 {
00035     Vertice *inicioGrafo;
00036 
00037 } Grafo;
00038 
00047 void ApagaGrafo(Grafo *g);
00048 
00058 Grafo *CriarGrafo(bool *inf);
00059 
00071 Grafo *InserirVerticeGrafo(Grafo *g, int novo, bool *inf);
00072 
00086 Grafo *InserirAdjGrafo(Grafo *g, int origem, int destino, int peso, bool *inf);
00087 
00099 Grafo *EliminaVerticeGrafo(Grafo *g, int id, bool *inf);
00100 
00113 Grafo *EliminaAdjGrafo(Grafo *g, int origem, int destino, bool *inf);
00114 
00126 bool ExisteAdjDoisVertices(Vertice *inicio, int origem, int destino);
00127 
00128 #endif
```

5.14 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_EDA_Hugo_Cruz_a23010/src/Main/libs/grafo.h File Reference

Ficheiro de cabeçalho para a estrutura de dados Grafo e funcionalidades.

```
#include "vertices.h"
```

Data Structures

- struct Grafo

  Estrutura de dados para um Grafo.

Macros

- #define MAX 10

  Define um valor.

Typedefs

- typedef struct Grafo Grafo

  Estrutura de dados para um Grafo.

Functions

- void ApagaGrafo (Grafo *g)

  Apaga um grafo e limpa a memória alocada.
- Grafo * CriarGrafo (bool *inf)

  Cria um novo grafo.
- Grafo * InserirVerticeGrafo (Grafo *g, int novo, bool *inf)

  Insere um novo vértice no grafo.
- Grafo * InserirAdjGrafo (Grafo *g, int origem, int destino, int peso, bool *inf)

  Insere uma nova aresta no grafo.
- Grafo * EliminaVerticeGrafo (Grafo *g, int id, bool *inf)

  Elimina um vértice do grafo.
- Grafo * EliminaAdjGrafo (Grafo *g, int origem, int destino, bool *inf)

  Elimina uma aresta do grafo.
- bool ExisteAdjDoisVertices (Vertice *inicio, int origem, int destino)

  Verifica se existe uma aresta entre dois vértices.

5.14.1 Detailed Description

Ficheiro de cabeçalho para a estrutura de dados Grafo e funcionalidades.

Author

Hugo Cruz (a23010)

Version

88.1

Date

2024-05-24

Copyright

Copyright (c) 2024

5.14.2 Macro Definition Documentation

5.14.2.1 MAX

```
#define MAX 10
```

Define um valor.
Esta macro define um valor maximo de 10.

5.14.3 Typedef Documentation

5.14.3.1 Grafo

```
typedef struct Grafo Grafo
```

Estrutura de dados para um Grafo.
A estrutura Grafo é uma representação de um grafo em memória, onde inicioGrafo é um apontador para o primeiro vértice do grafo.

5.14.4 Function Documentation

5.14.4.1 ApagaGrafo()

```
void ApagaGrafo (
            Grafo * g)
```

Apaga um grafo e limpa a memória alocada.
Esta função apaga um grafo e liberta a memória alocada. Ela percorre todos os vértices e adjacências do grafo, apagando-os um a um

Parameters

g	Apontador para o grafo a ser apagado

```
00022 {
00023     if (g == NULL)
00024         return;
00025
00026     Vertice* v = g->inicioGrafo;
00027
00028     //Apagar todos os vertices e adjacências
00029     while (v != NULL)
00030     {
00031         Adjacente* a = v->nextA;
00032         while (a != NULL)
00033         {
00034             Adjacente* temp = a;
00035             a = a->next;
00036             ApagarAdjacencia(temp);
00037         }
00038
00039         Vertice* tempV = v;
00040         v = v->nextV;
00041         ApagarVertice(tempV);
00042     }
00043
00044     free(g);
00045 }
```

5.14.4.2 CriarGrafo()

```
Grafo * CriarGrafo (
            bool * inf)
```

Cria um novo grafo.
Esta função cria um novo grafo. Ela aloca memória para um novo grafo e retorna um apontador para o grafo criado.

Parameters

inf	Apontador para uma variável booleana que será definida como true se a criação do grafo for bem-sucedida, e false caso contrário.

Returns

Apontador para o novo grafo criado.

```
00057 {
00058     *inf = false;
00059
00060     Grafo* aux = (Grafo*)malloc(sizeof(Grafo)); //Aloca memória para um grafo
00061
00062     if (aux == NULL)
00063     {
00064         *inf = false;
00065         return NULL;
00066     }
00067
00068     aux->inicioGrafo = NULL;
00069
00070     *inf = true;
00071     return aux;
00072 }
```

5.14.4.3 EliminaAdjGrafo()

```
Grafo * EliminaAdjGrafo (
            Grafo * g,
            int origem,
            int destino,
            bool * inf)
```

Elimina uma aresta do grafo.
Esta função elimina uma aresta do grafo. Ela verifica se a aresta a ser eliminada existe e, em seguida, remove-a do grafo.

Parameters

g	Apontador para o grafo do qual a aresta será eliminada.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.
inf	Apontador para uma variável booleana que será definida como true se a eliminação for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com a aresta eliminada.

```
00194 {
00195     *inf = false;
00196 
00197     if (g == NULL)
00198     {
00199         return NULL;
00200     }
00201 
00202     //Coloca se no posição de origem
00203     Vertice* origemVertice = ColocaNumaPosicaoLista(g->inicioGrafo, origem, &inf);
00204 
00205     if (origemVertice == NULL)
00206     {
00207         *inf = false;
00208         return g;
00209     }
00210 
00211     //Coloca se no posição de destino
00212     Vertice* destinoVertice = ColocaNumaPosicaoLista(g->inicioGrafo, destino, &inf);
00213 
00214     if (destinoVertice == NULL)
00215     {
00216         *inf = false;
00217         return g;
00218     }
00219 
00220     //Apaga um adjacência entre dois vértices
00221     origemVertice->nextA = EliminaUmaAdj(origemVertice->nextA, destino, &inf);
00222 
00223     return g;
00224 }
```

5.14.4.4 EliminaVerticeGrafo()

```
Grafo * EliminaVerticeGrafo (
            Grafo * g,
            int id,
            bool * inf)
```

Elimina um vértice do grafo.
Esta função elimina um vértice do grafo. Ela verifica se o vértice a ser eliminado existe e, em seguida, remove-o do grafo.

Parameters

g	Apontador para o grafo do qual o vértice será eliminado.
id	O valor do vértice a ser eliminado.

Parameters

inf	Apontador para uma variável booleana que será definida como true se a eliminação for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com o vértice eliminado.

```
00165 {
00166     *inf = false;
00167 
00168     if (g == NULL )
00169     {
00170         return NULL;
00171     }
00172 
00173     //Elimina o vértice da lista de vertices do grafo
00174     g->inicioGrafo = EliminarVertice(g->inicioGrafo, id, &inf);
00175     //Apaga todas as adjacências relacionadas com o id
00176     g->inicioGrafo = EliminarTodasAdjacenciasVertice(g->inicioGrafo, id, &inf);
00177 
00178     return g;
00179 }
```

5.14.4.5 ExisteAdjDoisVertices()

```
bool ExisteAdjDoisVertices (
            Vertice * inicio,
            int origem,
            int destino)
```

Verifica se existe uma aresta entre dois vértices.
Esta função verifica se existe uma aresta entre dois vértices. Ela percorre todos os vértices e adjacências do grafo, procurando uma aresta entre os vértices de origem e destino.

Parameters

inicio	Apontador para o primeiro vértice do grafo.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.

Returns

bool Retorna true se existe uma aresta entre os vértices origem e destino, e false caso contrário.

```
00238 {
00239     if (inicio == NULL )
00240     {
00241         return false;
00242     }
00243 
00244     if (ExisteVertice(inicio, origem))
00245     {
00246         Vertice* auxV = inicio;
00247 
00248         while (auxV)
00249         {
00250             Adjacente* auxA = auxV->nextA;
00251 
00252             while (auxA)
00253             {
00254                 //Procura dois vértices e  informa se os mesmos existem
00255                 if (auxV->id == origem && auxA->id == destino)
00256                 {
00257                     if (auxA->peso > 0)
00258                     {
00259                         return true;
00260                     }
00261                 }
00262 
00263                 auxA = auxA->next;
00264             }
00265 
```

```
00266             auxV = auxV->nextV;
00267         }
00268
00269         return false;
00270     }
00271     else
00272     {
00273         return false;
00274     }
00275 }
```

5.14.4.6 InserirAdjGrafo()

```
Grafo * InserirAdjGrafo (
            Grafo * g,
            int origem,
            int destino,
            int peso,
            bool * inf)
```

Insere uma nova aresta no grafo.
Esta função insere uma nova aresta no grafo. Ela verifica se os vértices de origem e destino são válidos e, em seguida, insere a aresta entre eles.

Parameters

g	Apontador para o grafo onde a aresta será inserida.
origem	O valor do vértice de origem da aresta.
destino	O valor do vértice de destino da aresta.
peso	O peso da aresta a ser inserida.
inf	Apontador para uma variável booleana que será definida como true se a inserção for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com a nova aresta inserida.

```
00121 {
00122     *inf = false;
00123
00124     if (g == NULL)
00125     {
00126         return NULL;
00127     }
00128
00129     //Coloca se no posição de origem
00130     Vertice* origemVertice = ColocaNumaPosicaoLista(g->inicioGrafo, origem, &inf);
00131
00132     if (origemVertice == NULL)
00133     {
00134         *inf = false;
00135         return g;
00136     }
00137
00138     //Coloca se no posição de destino
00139     Vertice* destinoVertice = ColocaNumaPosicaoLista(g->inicioGrafo, destino, &inf);
00140
00141     if (destinoVertice == NULL)
00142     {
00143         *inf = false;
00144         return g;
00145     }
00146
00147     //Insere uma adjacência
00148     origemVertice->nextA = InserirAdjacenciaLista(origemVertice->nextA, destino, peso, &inf);
00149
00150     return g;
00151 }
```

5.14.4.7 InserirVerticeGrafo()

```
Grafo * InserirVerticeGrafo (
            Grafo * g,
```

```
            int novo,
            bool * inf)
```

Insere um novo vértice no grafo.
Esta função insere um novo vértice no grafo. Ela verifica se o valor do novo vértice já existe e, em seguida, insere-o no fim da lista de vertices.

Parameters

g	Apontador para o grafo onde o vértice será inserido.
novo	O valor do novo vértice a ser inserido.
inf	Apontador para uma variável booleana que será definida como true se a inserção for bem-sucedida, e false caso contrário.

Returns

Apontador para o grafo com o novo vértice inserido.

```
00086 {
00087     *inf = false;
00088 
00089     if (g == NULL)
00090     {
00091         return NULL;
00092     }
00093     else
00094     {
00095         //Insere o novo vértice na lista de vertices do grafo
00096         g->inicioGrafo = InserirVerticeLista(g->inicioGrafo, novo, &inf);
00097 
00098         if (g->inicioGrafo != NULL)
00099         {
00100             *inf = true;
00101         }
00102     }
00103 
00104     return g;
00105 }
```

5.15 grafo.h

Go to the documentation of this file.

```
00001 
00012 #ifndef GRAFO_H
00013 #define GRAFO_H
00014 
00021 #define MAX 10
00022 
00023 #include "vertices.h"
00024 
00033 typedef struct Grafo
00034 {
00035     Vertice *inicioGrafo;
00036 
00037 } Grafo;
00038 
00047 void ApagaGrafo(Grafo *g);
00048 
00058 Grafo *CriarGrafo(bool *inf);
00059 
00071 Grafo *InserirVerticeGrafo(Grafo *g, int novo, bool *inf);
00072 
00086 Grafo *InserirAdjGrafo(Grafo *g, int origem, int destino, int peso, bool *inf);
00087 
00099 Grafo *EliminaVerticeGrafo(Grafo *g, int id, bool *inf);
00100 
00113 Grafo *EliminaAdjGrafo(Grafo *g, int origem, int destino, bool *inf);
00114 
00126 bool ExisteAdjDoisVertices(Vertice *inicio, int origem, int destino);
00127 
00128 #endif
```

5.16 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/InputOutput.c File Reference

Este ficheiro contém funções para carregar e mostrar dados. As funções de carregamento podem ler dados a partir de ficheiros.

```
#include "InputOutput.h"
```

Functions

- void MostraVertice (Vertice *grafo)

  Função para mostar vértices as adjacências.
- void MostraGrafo (Grafo *g)

  Função para mostra um grafo.
- char * ReadFile (char *file)

  Função para ler um arquivo.
- void Contador (char *dados, int *linha, int *coluna)

  Função para contar o número de linhas e colunas em uma string.
- Grafo * CriarVerticesCSV (char *dados)

  Função para criar vértices a partir de um arquivo CSV.
- Grafo * CarregaDadosCSV (char *file)

  Função para carregar dados de um arquivo CSV.
- void GuardaVertices (Grafo *g, char *file)

  Função para guardar vértices em um ficheiro binário.
- Grafo * CarregaVertices (char *file)

  Função para carregar vértices de um arquivo.
- void GuardarAdjacentes (Grafo *g, char *file)

  Função para guardar adjacências em um ficheiro binário.
- Grafo * CarregaAdjacencias (Grafo *grafo, char *file)

  Função para carregar adjacências de um arquivo num grafo.
- void GuardaGrafo (Grafo *g, char *vertices, char *adjacencias)

  Função para guardar um grafo em dois arquivos, um para vértices e outro para adjacências.
- Grafo * CarregaGrafo (char *vertices, char *adjacencias)

  Função para carregar um grafo a partir de dois arquivos, um para vértices e outro para adjacências.
- Grafo * CarregaDados (char *file, char *vertices, char *adjacencias)

  Função para carregar dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível.
- void ImprimirCaminho (int verticeAnt[ ], int destino)

  Mostra o caminho de um vértice de origem a um vértice de destino.
- void MostrarCaminho (Grafo *g, int origem, int destino)

  Mostra o caminho mais curto entre dois vértices num grafo.

5.16.1 Detailed Description

Este ficheiro contém funções para carregar e mostrar dados. As funções de carregamento podem ler dados a partir de ficheiros.

Author

Hugo Cruz (a23010)

Version

445.1

Date

2024-05-24

Copyright

5.16.2 Function Documentation

5.16.2.1 CarregaAdjacencias()

```
Grafo * CarregaAdjacencias (
            Grafo * grafo,
            char * file)
```

Função para carregar adjacências de um arquivo num grafo.
Esta função carrega as adjacências de um ficheiro binário para um grafo. Lê o ficheiro e adiciona cada adjacência ao grafo correspondente.

Parameters

grafo	Apontador para o grafo onde seram carregado os dados.
file	Nome do arquivo.

Returns

Grafo* Retorna um apontador para o grafo atualizado.

```
00361 {
00362     bool inf;
00363     FILE* ficheiro = fopen(file, "rb");
00364     if (ficheiro == NULL) return NULL;
00365 
00366     AdjacenteFile auxAF;
00367 
00368     //Le o cabeçalho
00369     fread(&auxAF, sizeof(auxAF), 1, ficheiro) == 1;
00370 
00371     //Se for o ficheiro com um id diferente de -8 não le
00372     if (auxAF.id != -8) return NULL;
00373 
00374     Vertice* aux = grafo->inicioGrafo;
00375 
00376 
00377     while (aux)
00378     {
00379         //Le até encontar o -1
00380         while (fread(&auxAF, sizeof(auxAF), 1, ficheiro) == 1 && auxAF.id != -1)
00381         {
00382             grafo = InserirAdjGrafo(grafo, aux->id, auxAF.id, auxAF.peso, &inf);
00383         }
00384         aux = aux->nextV;
00385     }
00386 
00387 
00388     fclose(ficheiro);
00389     return grafo;
00390 }
```

5.16.2.2 CarregaDados()

```
Grafo * CarregaDados (
            char * file,
            char * vertices,
            char * adjacencias)
```

Função para carregar dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível.
Esta função carrega dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível. Se os ficheiros de vértices e adjacências estiverem disponíveis, usa-os para carregar o grafo. Caso contrário, carrega os dados de um ficheiro CSV.

Parameters

file	Nome do arquivo com dados (argumento).
vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para vértices.

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00440 {
00441     bool inf;
00442
00443     Grafo* g = CriarGrafo(&inf);
00444
00445     g = CarregaGrafo(vertices, adjacencias);
00446
00447     //Se ocorrer algum erro com os ficheiros le o csv
00448     if (g != NULL)
00449     {
00450         return g;
00451     }
00452     else
00453     {
00454         g = CarregaDadosCSV(file);
00455         return g;
00456     }
00457
00458 }
```

5.16.2.3 CarregaDadosCSV()

```
Grafo * CarregaDadosCSV (
            char * file)
```

Função para carregar dados de um arquivo CSV.
Esta função recebe uma string de dados que representa um ficheiro CSV. Cria um grafo e insere vértices no grafo com base nos dados. O número de vértices inseridos é o maior entre o número de linhas e o número de colunas nos dados. Retorna um apontador para o grafo criado.

Parameters

file	Nome do arquivo CSV (argumento 1).

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00195 {
00196     bool inf;
00197
00198     int linhas = 0, colunas = 0;
00199     char* saveptr_linha = NULL;
00200     char* saveptr_coluna = NULL;
00201
00202     char* dados = ReadFile(file); //Le tudo do ficheiro csv
00203     char* dados_copy = strdup(dados); // Cria uma c♦pia da string lida
00204     Grafo* g = CriarVerticesCSV(dados_copy); // Conta e cria os vertices
00205
00206     //Divide uma string em tokens separados por \n
00207     dados = strtok_s(dados, "\n", &saveptr_linha);
00208
00209     while (dados != NULL)
00210     {
00211         linhas++;
```

```
00212         dados = strtok_s(dados, ";", &saveptr_coluna); //Divide uma string em tokens separados por ;
00213
00214         while (dados != NULL)
00215         {
00216             colunas++;
00217             g = InserirAdjGrafo(g, linhas, colunas, atoi(dados), &inf); //Tranforma uma string em
      inteiro
00218             dados = strtok_s(NULL, ";", &saveptr_coluna); //Coloca NULL e avança
00219         }
00220         colunas = 0;
00221         dados = strtok_s(NULL, "\n", &saveptr_linha);
00222     }
00223
00224     free(dados);
00225     return g;
00226
00227 }
```

5.16.2.4 CarregaGrafo()

```
Grafo * CarregaGrafo (
            char * vertices,
            char * adjacencias)
```

Função para carregar um grafo a partir de dois arquivos, um para vértices e outro para adjacências.
Esta função carrega um grafo inteiro a partir de dois ficheiros, um para os vértices e outro para as adjacências.
Usa as funções CarregaVertices e CarregaAdjacencias

Parameters

vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para adjacências.

Returns

Grafo* Retorna um apontador para o grafo com os valores carregados

```
00419 {
00420     Grafo* g = CarregaVertices(vertices);
00421     if (g == NULL)return NULL;
00422     Grafo* grafo = CarregaAdjacencias(g, adjacencias);
00423
00424     return grafo;
00425 }
```

5.16.2.5 CarregaVertices()

```
Grafo * CarregaVertices (
            char * file)
```

Função para carregar vértices de um arquivo.
Esta função carrega os vértices de um ficheiro binário para um grafo. Lê o ficheiro e adiciona cada vértice ao grafo.

Parameters

file	Nome do arquivo

Returns

Grafo* Retorna um apontador para o grafo criado

```
00272 {
00273     bool inf;
00274     FILE* ficheiro = fopen(file, "rb");
00275
00276     if (ficheiro == NULL) return;
00277
00278     Grafo* g = CriarGrafo(&inf);//Cria um grafo
00279
00280     VerticeFile auxVF;
00281     Vertice* novo = NULL;
00282
```

```
00283     //Le o cabeçalho
00284     fread(&auxVF, sizeof(auxVF), 1, ficheiro) == 1;
00285
00286     //Se for o ficheiro correto contínua a ler
00287     if (auxVF.id == -7)
00288     {
00289         while (fread(&auxVF, sizeof(auxVF), 1, ficheiro) == 1)
00290         {
00291             g = InserirVerticeGrafo(g, auxVF.id, &inf);
00292         }
00293     }
00294     else
00295     {
00296         return NULL;
00297     }
00298
00299     fclose(ficheiro);//Fecha o ficheiro
00300     return g;
00301 }
```

5.16.2.6 Contador()

```
void Contador (
            char * dados,
            int * linha,
            int * coluna)
```

Função para contar o número de linhas e colunas em uma string.
Esta função recebe uma string de dados e dois apontadores para inteiros. Conta o número de linhas e colunas na string de dados e armazena esses valores nos inteiros apontados pelos apontadores.

Parameters

dados	Dados a serem contados.
linha	Apontador para o número de linhas.
coluna	Apontador para o número de colunas.

```
00103 {
00104     char* saveptr_linha = NULL;
00105     char* saveptr_coluna = NULL;
00106     int colunasNaLinha = 0;
00107     *linha = 0;
00108     *coluna = 0;
00109
00110     //Divide uma string em tokens separados por \n
00111     dados = strtok_s(dados, "\n", &saveptr_linha);
00112
00113     while (dados != NULL)
00114     {
00115         (*linha)++;
00116
00117         //Divide uma string em tokens separados por ;
00118         dados = strtok_s(dados, ";", &saveptr_coluna);
00119
00120         while (dados != NULL)
00121         {
00122             colunasNaLinha++;
00123             //Coloca NULL e avança
00124             dados = strtok_s(NULL, ";", &saveptr_coluna);
00125         }
00126
00127         //Guarda o número maior de linhas contadas
00128         if (colunasNaLinha > *coluna)
00129         {
00130             *coluna = colunasNaLinha;
00131         }
00132
00133         colunasNaLinha = 0;
00134
00135         dados = strtok_s(NULL, "\n", &saveptr_linha);
00136     }
00137
00138 }
```

5.16.2.7 CriarVerticesCSV()

```
Grafo * CriarVerticesCSV (
            char * dados)
```

Função para criar vértices a partir de um arquivo CSV.
Esta função conta as linhas e colunas de um token e cria memória para um grafo. Após a criação do mesmo carrega os vertices de um ficheiro CSV.

Parameters

dados	Dados do arquivo CSV.

Returns

Grafo∗ Retorna um apontador para o grafo criado.

```
00150 {
00151     bool inf;
00152     int linhas = 0;
00153     int colunas = 0;
00154
00155     //Conta linhas e colunas
00156     Contador(dados, &linhas, &colunas);
00157
00158     Grafo* g = CriarGrafo(&inf);
00159
00160     if (linhas == NULL || colunas == NULL) return NULL;
00161     //Cria o número de vertices que corresponde ao maior valor
00162     if (linhas <= colunas)
00163     {
00164         for (int i = 1; i <= colunas; i++)
00165         {
00166             g = InserirVerticeGrafo(g, i, &inf);
00167
00168         }
00169     }
00170     else
00171     {
00172         for (int i = 1; i <= linhas; i++)
00173         {
00174             g = InserirVerticeGrafo(g, i, &inf);
00175
00176         }
00177     }
00178
00179     free(dados);
00180     return g;
00181
00182 }
```

5.16.2.8 GuardaGrafo()

```
void GuardaGrafo (
            Grafo * g,
            char * vertices,
            char * adjacencias)
```

Função para guardar um grafo em dois arquivos, um para vértices e outro para adjacências.
Esta função guarda um grafo inteiro em dois ficheiros, um para os vértices e outro para as adjacências. Usa as funções GuardaVertices e GuardarAdjacentes

Parameters

g	Apontador para o grafo a guardar.
vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para adjacências.

```
00403 {
00404     GuardaVertices(g, vertices);
00405     GuardarAdjacentes(g, adjacencias);
00406 }
```

5.16.2.9 GuardarAdjacentes()

```
void GuardarAdjacentes (
            Grafo * g,
            char * file)
```

Função para guardar adjacências em um ficheiro binário.
Esta função guarda as adjacências de um grafo num ficheiro binário. Percorre todos os vértices e as suas adjacências e escreve-os no ficheiro.

Parameters

g	Apontador para o grafo a guardar
file	Nome do arquivo

```
00313 {
00314     FILE* ficheiro = fopen(file, "wb");
00315
00316     if (ficheiro == NULL) return;
00317
00318     Vertice* auxV = g->inicioGrafo;
00319     AdjacenteFile auxAF; // Estrutura sem apontadores
00320
00321     //Adicina um cabeçalho para garantir que le o ficheiro correto
00322     auxAF.id = -8;
00323     fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00324
00325     //Avança com os vertices, mas apenas escreve as adjacências
00326     while (auxV)
00327     {
00328         Adjacente* auxA = auxV->nextA;
00329
00330         while (auxA)
00331         {
00332             auxAF.id = auxA->id;
00333             auxAF.peso = auxA->peso;
00334
00335             fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00336
00337             auxA = auxA->next;
00338         }
00339
00340         //Marca para sair do while na leitura
00341         auxAF.id = -1;
00342         fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00343
00344         auxV = auxV->nextV;
00345     }
00346
00347     fclose(ficheiro);
00348 }
```

5.16.2.10 GuardaVertices()

```
void GuardaVertices (
            Grafo * g,
            char * file)
```

Função para guardar vértices em um ficheiro binário.
Esta função guarda os vértices de um grafo num ficheiro binário. Percorre todos os vértices do grafo e escreve-os no ficheiro.

Parameters

g	Apontador para o grafo a guardar.
file	Nome do arquivo.

```
00239 {
00240     FILE* ficheiro = fopen(file, "wb");
00241
00242     if (ficheiro == NULL) return;
00243
00244     Vertice* auxV = g->inicioGrafo;
00245     VerticeFile auxVF; // Estrutura sem apontadores
00246
00247     //Adicina um cabeçalho para garantir que le o ficheiro correto
00248     auxVF.id = -7;
00249     fwrite(&auxVF, sizeof(VerticeFile), 1, ficheiro);
00250
00251     //Escreve todos os vertices em modo binário
00252     while (auxV)
00253     {
```

```
00254         auxVF.id = auxV->id;
00255         fwrite(&auxVF, sizeof(VerticeFile), 1, ficheiro);
00256         auxV = auxV->nextV;
00257     }
00258
00259     fclose(ficheiro);  // Fecha o ficheiro após a escrita
00260 }
```

5.16.2.11 ImprimirCaminho()

```
void ImprimirCaminho (
            int verticeAnt[],
            int destino)
```

Mostra o caminho de um vértice de origem a um vértice de destino.

Parameters

verticeAnt	Array que contém os antecessore de cada vertice
destino	Destino final

```
00467 {
00468     int caminho[MAX];
00469     int contador = 0;
00470     int atual = destino;
00471
00472      // Coloca num arrays todos os anteriores a cada vertice
00473     while (atual != -1)
00474     {
00475         caminho[contador++] = atual;
00476         atual = verticeAnt[atual];
00477     }
00478
00479     // Imprime o caminho do vértice de origem ao vértice de destino
00480     for (int i = contador - 1; i >= 0; i--)
00481     {
00482         printf("%d ", caminho[i]);
00483     }
00484 }
```

5.16.2.12 MostraGrafo()

```
void MostraGrafo (
            Grafo * g)
```

Função para mostra um grafo.
Esta função recebe um apontador para um grafo e imprime o grafo chamando a função MostraVertice para o vértice inicial do grafo.

Parameters

g	Apontador para o grafo a ser mostrado

```
00054 {
00055     MostraVertice(g->inicioGrafo);
00056 }
```

5.16.2.13 MostrarCaminho()

```
void MostrarCaminho (
            Grafo * g,
            int origem,
            int destino)
```

Mostra o caminho mais curto entre dois vértices num grafo.
Esta função recebe um grafo, um vértice de origem e um vértice de destino. Ela utiliza o algoritmo de Dijkstra para calcular o caminho mais curto do vértice de origem ao vértice de destino e imprime esse caminho.

Parameters

g	O grafo.

Parameters

origem	O vértice de origem.
destino	O vértice de destino.

```
00497 {
00498     if (g == NULL)
00499     {
00500         return;
00501     }
00502
00503     int distanciasFinais[MAX];
00504     int verticeAnt[MAX];
00505
00506     // Inicializa o array de vértices anteriores
00507     for (int i = 0; i < MAX; i++)
00508     {
00509         verticeAnt[i] = -1;
00510     }
00511
00512     // Inicializa o array de vértices anteriores
00513     InicializarArrays(distanciasFinais);
00514
00515     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00516
00517     if (distanciasFinais[destino] == INT_MAX)
00518     {
00519         printf("Não existe caminho de %d para %d\n", origem, destino);
00520     }
00521     else
00522     {
00523         printf("Caminho de %d para %d: ", origem, destino);
00524         ImprimirCaminho(verticeAnt, destino);
00525         printf("\nDistância: %d\n", distanciasFinais[destino]);
00526     }
00527 }
```

5.16.2.14 MostraVertice()

```
void MostraVertice (
            Vertice * grafo)
```

Função para mostar vértices as adjacências.
Função para mostar vértices as adjacencias.
Esta função recebe um apontador para um vértice e imprime o vértice e as suas adjacências. Percorre a lista de vértices e para cada vértice, percorre a lista de adjacências, imprimindo as.

Parameters

grafo	Apontador para o vértice a ser mostrado.

```
00021 {
00022     Vertice* aux = grafo;
00023
00024     //Avança na lista de vertices e arestas e mostra todos os pesos das adjacência
00025     while (aux != NULL)
00026     {
00027         printf("\nVértice: %d\n\n", aux->id);
00028
00029         Adjacente* adj = aux->nextA;
00030
00031         while (adj)
00032         {
00033             if (adj->peso > 0 && adj->peso != INT_MAX)
00034             {
00035                 printf("\tAdj: %d - (%d)\n", adj->id, adj->peso);
00036
00037             }
00038             adj = adj->next;
00039         }
00040
00041         aux = aux->nextV;
00042     }
00043     printf("\n");
00044 }
```

5.16.2.15 ReadFile()

```
char * ReadFile (
            char * file)
```

Função para ler um arquivo.
Esta função recebe o nome de um ficheiro, lê o ficheiro e retorna os dados lidos. Se o ficheiro não puder ser aberto, retorna NULL. Caso contrário, lê o ficheiro inteiro para uma string e retorna essa string.

Parameters

file	Nome do arquivo a ser lido.

Returns

char* Retorna os dados lidos do arquivo.

```
00067 {
00068     FILE* ficheiro = fopen(file, "r");
00069     if (ficheiro == NULL) return NULL;
00070
00071     fseek(ficheiro, 0, SEEK_END); //Coloca o apontador para fille no fim do arquivo
00072     int tamanho = ftell(ficheiro); //Calcula o tamanho em bytes
00073     fseek(ficheiro, 0, SEEK_SET); //Coloca o apontador para fille no início do arquivo
00074
00075     char* dados = (char*)malloc(sizeof(char) * tamanho + 1); //Aloca memória para o tamanho do
      ficheiro + 1
00076
00077     //Informa me o tamanho lido em bytes coloca o final da string como NULL
00078     if (dados != NULL)
00079     {
00080         size_t bytesRead = fread(dados, 1, tamanho, ficheiro);
00081         dados[bytesRead] = '\0';
00082     }
00083     else
00084     {
00085         free(dados);
00086         dados = NULL;
00087     }
00088
00089     fclose(ficheiro);
00090     return dados;
00091 }
```

5.17 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/InputOutput.h File Reference

Este ficheiro de cabeçalho define as funções para carregar e mostrar dados a partir de ficheiros.

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include "grafo.h"
```

Functions

- void MostraVertice (Vertice *grafo)

  Função para mostar vértices as adjacencias.
- void MostraGrafo (Grafo *g)

  Função para mostra um grafo.
- char * ReadFile (char *file)

  Função para ler um arquivo.
- void Contador (char *dados, int *linha, int *coluna)

  Função para contar o número de linhas e colunas em uma string.
- Grafo * CriarVerticesCSV (char *dados)

Função para criar vértices a partir de um arquivo CSV.

- Grafo ∗ CarregaDadosCSV (char ∗file)

  Função para carregar dados de um arquivo CSV.
- void GuardaVertices (Grafo ∗g, char ∗file)

  Função para guardar vértices em um ficheiro binário.
- Grafo ∗ CarregaVertices (char ∗file)

  Função para carregar vértices de um arquivo.
- void GuardarAdjacentes (Grafo ∗g, char ∗file)

  Função para guardar adjacências em um ficheiro binário.
- Grafo ∗ CarregaAdjacencias (Grafo ∗grafo, char ∗file)

  Função para carregar adjacências de um arquivo num grafo.
- void GuardaGrafo (Grafo ∗g, char ∗vertices, char ∗adjacencias)

  Função para guardar um grafo em dois arquivos, um para vértices e outro para adjacências.
- Grafo ∗ CarregaGrafo (char ∗vertices, char ∗adjacencias)

  Função para carregar um grafo a partir de dois arquivos, um para vértices e outro para adjacências.
- Grafo ∗ CarregaDados (char ∗file, char ∗vertices, char ∗adjacencias)

  Função para carregar dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível.
- void ImprimirCaminho (int verticeAnt[ ], int destino)

  Mostra o caminho de um vértice de origem a um vértice de destino.
- void MostrarCaminho (Grafo ∗g, int origem, int destino)

  Mostra o caminho mais curto entre dois vértices num grafo.

5.17.1 Detailed Description

Este ficheiro de cabeçalho define as funções para carregar e mostrar dados a partir de ficheiros.

Author

Hugo Cruz (a23010)

Version

71.1

Date

2024-05-24

Copyright

Copyright (c) 2024

5.17.2 Function Documentation

5.17.2.1 CarregaAdjacencias()

```
Grafo * CarregaAdjacencias (
            Grafo * grafo,
            char * file)
```

Função para carregar adjacências de um arquivo num grafo.
Esta função carrega as adjacências de um ficheiro binário para um grafo. Lê o ficheiro e adiciona cada adjacência ao grafo correspondente.

Parameters

grafo	Apontador para o grafo onde seram carregado os dados.
file	Nome do arquivo.

Returns

Grafo* Retorna um apontador para o grafo atualizado.

```
00361 {
00362     bool inf;
00363     FILE* ficheiro = fopen(file, "rb");
00364     if (ficheiro == NULL) return NULL;
00365
00366     AdjacenteFile auxAF;
00367
00368     //Le o cabeçalho
00369     fread(&auxAF, sizeof(auxAF), 1, ficheiro) == 1;
00370
00371     //Se for o ficheiro com um id diferente de -8 não le
00372     if (auxAF.id != -8) return NULL;
00373
00374     Vertice* aux = grafo->inicioGrafo;
00375
00376
00377     while (aux)
00378     {
00379         //Le até encontar o -1
00380         while (fread(&auxAF, sizeof(auxAF), 1, ficheiro) == 1 && auxAF.id != -1)
00381         {
00382             grafo = InserirAdjGrafo(grafo, aux->id, auxAF.id, auxAF.peso, &inf);
00383         }
00384         aux = aux->nextV;
00385     }
00386
00387
00388     fclose(ficheiro);
00389     return grafo;
00390 }
```

5.17.2.2 CarregaDados()

```
Grafo * CarregaDados (
            char * file,
            char * vertices,
            char * adjacencias)
```

Função para carregar dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível.

Esta função carrega dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível. Se os ficheiros de vértices e adjacências estiverem disponíveis, usa-os para carregar o grafo. Caso contrário, carrega os dados de um ficheiro CSV.

Parameters

file	Nome do arquivo com dados (argumento).
vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para vértices.

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00440 {
00441     bool inf;
00442
00443     Grafo* g = CriarGrafo(&inf);
00444
00445     g = CarregaGrafo(vertices, adjacencias);
00446
00447     //Se ocorrer algum erro com os ficheiros le o csv
00448     if (g != NULL)
00449     {
00450         return g;
00451     }
00452     else
00453     {
00454         g = CarregaDadosCSV(file);
00455         return g;
00456     }
00457
00458 }
```

5.17.2.3 CarregaDadosCSV()

```
Grafo * CarregaDadosCSV (
            char * file)
```

Função para carregar dados de um arquivo CSV.
Esta função recebe uma string de dados que representa um ficheiro CSV. Cria um grafo e insere vértices no grafo com base nos dados. O número de vértices inseridos é o maior entre o número de linhas e o número de colunas nos dados. Retorna um apontador para o grafo criado.

Parameters

file	Nome do arquivo CSV (argumento 1).

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00195 {
00196     bool inf;
00197 
00198     int linhas = 0, colunas = 0;
00199     char* saveptr_linha = NULL;
00200     char* saveptr_coluna = NULL;
00201 
00202     char* dados = ReadFile(file); //Le tudo do ficheiro csv
00203     char* dados_copy = strdup(dados); // Cria uma c♦pia da string lida
00204     Grafo* g = CriarVerticesCSV(dados_copy); // Conta e cria os vertices
00205 
00206     //Divide uma string em tokens separados por \n
00207     dados = strtok_s(dados, "\n", &saveptr_linha);
00208 
00209     while (dados != NULL)
00210     {
00211         linhas++;
00212         dados = strtok_s(dados, ";", &saveptr_coluna); //Divide uma string em tokens separados por ;
00213 
00214         while (dados != NULL)
00215         {
00216             colunas++;
00217             g = InserirAdjGrafo(g, linhas, colunas, atoi(dados), &inf); //Tranforma uma string em
      inteiro
00218             dados = strtok_s(NULL, ";", &saveptr_coluna); //Coloca NULL e avança
00219         }
00220         colunas = 0;
00221         dados = strtok_s(NULL, "\n", &saveptr_linha);
00222     }
00223 
00224     free(dados);
00225     return g;
00226 
00227 }
```

5.17.2.4 CarregaGrafo()

```
Grafo * CarregaGrafo (
            char * vertices,
            char * adjacencias)
```

Função para carregar um grafo a partir de dois arquivos, um para vértices e outro para adjacências.
Esta função carrega um grafo inteiro a partir de dois ficheiros, um para os vértices e outro para as adjacências. Usa as funções CarregaVertices e CarregaAdjacencias

Parameters

vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para adjacências.

Returns

Grafo* Retorna um apontador para o grafo com os valores carregados

```
00419 {
00420     Grafo* g = CarregaVertices(vertices);
00421     if (g == NULL)return NULL;
00422     Grafo* grafo = CarregaAdjacencias(g, adjacencias);
00423 
00424     return grafo;
00425 }
```

5.17.2.5 CarregaVertices()

```
Grafo * CarregaVertices (
            char * file)
```

Função para carregar vértices de um arquivo.
Esta função carrega os vértices de um ficheiro binário para um grafo. Lê o ficheiro e adiciona cada vértice ao grafo.

Parameters

file	Nome do arquivo

Returns

Grafo* Retorna um apontador para o grafo criado

```
00272 {
00273     bool inf;
00274     FILE* ficheiro = fopen(file, "rb");
00275 
00276     if (ficheiro == NULL) return;
00277 
00278     Grafo* g = CriarGrafo(&inf);//Cria um grafo
00279 
00280     VerticeFile auxVF;
00281     Vertice* novo = NULL;
00282 
00283     //Le o cabeçalho
00284     fread(&auxVF, sizeof(auxVF), 1, ficheiro) == 1;
00285 
00286     //Se for o ficheiro correto contínua a ler
00287     if (auxVF.id == -7)
00288     {
00289         while (fread(&auxVF, sizeof(auxVF), 1, ficheiro) == 1)
00290         {
00291             g = InserirVerticeGrafo(g, auxVF.id, &inf);
00292         }
00293     }
00294     else
00295     {
00296         return NULL;
00297     }
00298 
00299     fclose(ficheiro);//Fecha o ficheiro
00300     return g;
00301 }
```

5.17.2.6 Contador()

```
void Contador (
            char * dados,
            int * linha,
            int * coluna)
```

Função para contar o número de linhas e colunas em uma string.
Esta função recebe uma string de dados e dois apontadores para inteiros. Conta o número de linhas e colunas na string de dados e armazena esses valores nos inteiros apontados pelos apontadores.

Parameters

dados	Dados a serem contados.
linha	Apontador para o número de linhas.

coluna	Apontador para o número de colunas.

```
00103 {
00104     char* saveptr_linha = NULL;
00105     char* saveptr_coluna = NULL;
00106     int colunasNaLinha = 0;
00107     *linha = 0;
00108     *coluna = 0;
00109 
00110     //Divide uma string em tokens separados por \n
00111     dados = strtok_s(dados, "\n", &saveptr_linha);
00112 
00113     while (dados != NULL)
00114     {
00115         (*linha)++;
00116 
00117         //Divide uma string em tokens separados por ;
00118         dados = strtok_s(dados, ";", &saveptr_coluna);
00119 
00120         while (dados != NULL)
00121         {
00122             colunasNaLinha++;
00123             //Coloca NULL e avança
00124             dados = strtok_s(NULL, ";", &saveptr_coluna);
00125         }
00126 
00127         //Guarda o número maior de linhas contadas
00128         if (colunasNaLinha > *coluna)
00129         {
00130             *coluna = colunasNaLinha;
00131         }
00132 
00133         colunasNaLinha = 0;
00134 
00135         dados = strtok_s(NULL, "\n", &saveptr_linha);
00136     }
00137 
00138 }
```

5.17.2.7 CriarVerticesCSV()

```
Grafo * CriarVerticesCSV (
            char * dados)
```

Função para criar vértices a partir de um arquivo CSV.
Esta função conta as linhas e colunas de um token e cria memória para um grafo. Após a criação do mesmo carrega os vertices de um ficheiro CSV.

Parameters

dados	Dados do arquivo CSV.

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00150 {
00151     bool inf;
00152     int linhas = 0;
00153     int colunas = 0;
00154 
00155     //Conta linhas e colunas
00156     Contador(dados, &linhas, &colunas);
00157 
00158     Grafo* g = CriarGrafo(&inf);
00159 
00160     if (linhas == NULL || colunas == NULL) return NULL;
00161     //Cria o número de vertices que corresponde ao maior valor
00162     if (linhas <= colunas)
00163     {
00164         for (int i = 1; i <= colunas; i++)
00165         {
00166             g = InserirVerticeGrafo(g, i, &inf);
00167 
00168         }
00169     }
00170     else
```

```
00171     {
00172         for (int i = 1; i <= linhas; i++)
00173         {
00174             g = InserirVerticeGrafo(g, i, &inf);
00175 
00176         }
00177     }
00178 
00179     free(dados);
00180     return g;
00181 
00182 }
```

5.17.2.8 GuardaGrafo()

```
void GuardaGrafo (
            Grafo * g,
            char * vertices,
            char * adjacencias)
```

Função para guardar um grafo em dois arquivos, um para vértices e outro para adjacências.
Esta função guarda um grafo inteiro em dois ficheiros, um para os vértices e outro para as adjacências. Usa as funções GuardaVertices e GuardarAdjacentes

Parameters

g	Apontador para o grafo a guardar.
vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para adjacências.

```
00403 {
00404     GuardaVertices(g, vertices);
00405     GuardarAdjacentes(g, adjacencias);
00406 }
```

5.17.2.9 GuardarAdjacentes()

```
void GuardarAdjacentes (
            Grafo * g,
            char * file)
```

Função para guardar adjacências em um ficheiro binário.
Esta função guarda as adjacências de um grafo num ficheiro binário. Percorre todos os vértices e as suas adjacências e escreve-os no ficheiro.

Parameters

g	Apontador para o grafo a guardar
file	Nome do arquivo

```
00313 {
00314     FILE* ficheiro = fopen(file, "wb");
00315 
00316     if (ficheiro == NULL) return;
00317 
00318     Vertice* auxV = g->inicioGrafo;
00319     AdjacenteFile auxAF; // Estrutura sem apontadores
00320 
00321     //Adicina um cabeçalho para garantir que le o ficheiro correto
00322     auxAF.id = -8;
00323     fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00324 
00325     //Avança com os vertices, mas apenas escreve as adjacências
00326     while (auxV)
00327     {
00328         Adjacente* auxA = auxV->nextA;
00329 
00330         while (auxA)
00331         {
00332             auxAF.id = auxA->id;
00333             auxAF.peso = auxA->peso;
00334 
```

```
00335             fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00336
00337             auxA = auxA->next;
00338         }
00339
00340         //Marca para sair do while na leitura
00341         auxAF.id = -1;
00342         fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00343
00344         auxV = auxV->nextV;
00345     }
00346
00347     fclose(ficheiro);
00348 }
```

5.17.2.10 GuardaVertices()

```
void GuardaVertices (
            Grafo * g,
            char * file)
```

Função para guardar vértices em um ficheiro binário.
Esta função guarda os vértices de um grafo num ficheiro binário. Percorre todos os vértices do grafo e escreve-os no ficheiro.

Parameters

g	Apontador para o grafo a guardar.
file	Nome do arquivo.

```
00239 {
00240     FILE* ficheiro = fopen(file, "wb");
00241
00242     if (ficheiro == NULL) return;
00243
00244     Vertice* auxV = g->inicioGrafo;
00245     VerticeFile auxVF; // Estrutura sem apontadores
00246
00247     //Adicina um cabeçalho para garantir que le o ficheiro correto
00248     auxVF.id = -7;
00249     fwrite(&auxVF, sizeof(VerticeFile), 1, ficheiro);
00250
00251     //Escreve todos os vertices em modo binário
00252     while (auxV)
00253     {
00254         auxVF.id = auxV->id;
00255         fwrite(&auxVF, sizeof(VerticeFile), 1, ficheiro);
00256         auxV = auxV->nextV;
00257     }
00258
00259     fclose(ficheiro);  // Fecha o ficheiro após a escrita
00260 }
```

5.17.2.11 ImprimirCaminho()

```
void ImprimirCaminho (
            int verticeAnt[],
            int destino)
```

Mostra o caminho de um vértice de origem a um vértice de destino.

Parameters

verticeAnt	Array que contém os antecessore de cada vertice
destino	Destino final

```
00467 {
00468     int caminho[MAX];
00469     int contador = 0;
00470     int atual = destino;
00471
00472      // Coloca num arrays todos os anteriores a cada vertice
00473     while (atual != -1)
00474     {
00475         caminho[contador++] = atual;
```

```
00476         atual = verticeAnt[atual];
00477     }
00478
00479     // Imprime o caminho do vértice de origem ao vértice de destino
00480     for (int i = contador - 1; i >= 0; i--)
00481     {
00482         printf("%d ", caminho[i]);
00483     }
00484 }
```

5.17.2.12 MostraGrafo()

```
void MostraGrafo (
            Grafo * g)
```

Função para mostra um grafo.
Esta função recebe um apontador para um grafo e imprime o grafo chamando a função MostraVertice para o vértice inicial do grafo.

Parameters

g	Apontador para o grafo a ser mostrado

```
00054 {
00055     MostraVertice(g->inicioGrafo);
00056 }
```

5.17.2.13 MostrarCaminho()

```
void MostrarCaminho (
            Grafo * g,
            int origem,
            int destino)
```

Mostra o caminho mais curto entre dois vértices num grafo.
Esta função recebe um grafo, um vértice de origem e um vértice de destino. Ela utiliza o algoritmo de Dijkstra para calcular o caminho mais curto do vértice de origem ao vértice de destino e imprime esse caminho.

Parameters

g	O grafo.
origem	O vértice de origem.
destino	O vértice de destino.

```
00497 {
00498     if (g == NULL)
00499     {
00500         return;
00501     }
00502
00503     int distanciasFinais[MAX];
00504     int verticeAnt[MAX];
00505
00506     // Inicializa o array de vértices anteriores
00507     for (int i = 0; i < MAX; i++)
00508     {
00509         verticeAnt[i] = -1;
00510     }
00511
00512     // Inicializa o array de vértices anteriores
00513     InicializarArrays(distanciasFinais);
00514
00515     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00516
00517     if (distanciasFinais[destino] == INT_MAX)
00518     {
00519         printf("Não existe caminho de %d para %d\n", origem, destino);
00520     }
00521     else
00522     {
00523         printf("Caminho de %d para %d: ", origem, destino);
00524         ImprimirCaminho(verticeAnt, destino);
00525         printf("\nDistância: %d\n", distanciasFinais[destino]);
00526     }
00527 }
```

5.17.2.14 MostraVertice()

```
void MostraVertice (
            Vertice * grafo)
```

Função para mostar vértices as adjacencias.
Esta função recebe um apontador para um vértice e imprime o vértice e as suas adjacências. Percorre a lista de vértices e para cada vértice, percorre a lista de adjacências, imprimindo as.

Parameters

grafo	Apontador para o vértice a ser mostrado.

Função para mostar vértices as adjacencias.
Esta função recebe um apontador para um vértice e imprime o vértice e as suas adjacências. Percorre a lista de vértices e para cada vértice, percorre a lista de adjacências, imprimindo as.

Parameters

grafo	Apontador para o vértice a ser mostrado.

```
00021 {
00022     Vertice* aux = grafo;
00023 
00024     //Avança na lista de vertices e arestas e mostra todos os pesos das adjacência
00025     while (aux != NULL)
00026     {
00027         printf("\nVértice: %d\n\n", aux->id);
00028 
00029         Adjacente* adj = aux->nextA;
00030 
00031         while (adj)
00032         {
00033             if (adj->peso > 0 && adj->peso != INT_MAX)
00034             {
00035                 printf("\tAdj: %d - (%d)\n", adj->id, adj->peso);
00036 
00037             }
00038             adj = adj->next;
00039         }
00040 
00041         aux = aux->nextV;
00042     }
00043     printf("\n");
00044 }
```

5.17.2.15 ReadFile()

```
char * ReadFile (
            char * file)
```

Função para ler um arquivo.
Esta função recebe o nome de um ficheiro, lê o ficheiro e retorna os dados lidos. Se o ficheiro não puder ser aberto, retorna NULL. Caso contrário, lê o ficheiro inteiro para uma string e retorna essa string.

Parameters

file	Nome do arquivo a ser lido.

Returns

char* Retorna os dados lidos do arquivo.

```
00067 {
00068     FILE* ficheiro = fopen(file, "r");
00069     if (ficheiro == NULL) return NULL;
00070 
00071     fseek(ficheiro, 0, SEEK_END); //Coloca o apontador para fille no fim do arquivo
00072     int tamanho = ftell(ficheiro); //Calcula o tamanho em bytes
00073     fseek(ficheiro, 0, SEEK_SET); //Coloca o apontador para fille no início do arquivo
00074 
```

```
00075     char* dados = (char*)malloc(sizeof(char) * tamanho + 1); //Aloca memória para o tamanho do
      ficheiro + 1
00076
00077     //Informa me o tamanho lido em bytes coloca o final da string como NULL
00078     if (dados != NULL)
00079     {
00080         size_t bytesRead = fread(dados, 1, tamanho, ficheiro);
00081         dados[bytesRead] = '\0';
00082     }
00083     else
00084     {
00085         free(dados);
00086         dados = NULL;
00087     }
00088
00089     fclose(ficheiro);
00090     return dados;
00091 }
```

5.18 InputOutput.h

Go to the documentation of this file.

```
00001
00011 #ifndef INPUTOUTPUT_H
00012 #define INPUTOUTPUT_H
00013
00014 #include <stdio.h>
00015 #include <string.h>
00016 #include <locale.h>
00017 #include "grafo.h"
00018
00019
00027 void MostraVertice(Vertice* grafo);
00028
00036 void MostraGrafo(Grafo* g);
00037
00038
00047 char* ReadFile(char* file);
00048
00058 void Contador(char* dados, int* linha, int* coluna);
00059
00069 Grafo* CriarVerticesCSV(char* dados);
00070
00081 Grafo* CarregaDadosCSV(char* file);
00082
00092 void GuardaVertices(Grafo* g, char* file);
00093
00103 Grafo* CarregaVertices(char* file);
00104
00105
00115 void GuardarAdjacentes(Grafo* g, char* file);
00126 Grafo* CarregaAdjacencias(Grafo* grafo, char* file);
00127
00128
00139 void GuardaGrafo(Grafo* g, char* vertices, char* adjacencias);
00150 Grafo* CarregaGrafo(char* vertices, char* adjacencias);
00163 Grafo* CarregaDados(char* file, char* vertices, char* adjacencias);
00164
00171 void ImprimirCaminho(int verticeAnt[], int destino);
00172
00183 void MostrarCaminho(Grafo* g, int origem, int destino);
00184
00185
00186
00187
00188 #endif
```

5.19 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Main/libs/InputOutput.h File Reference

Este ficheiro de cabeçalho define as funções para carregar e mostrar dados a partir de ficheiros.

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include "grafo.h"
```

Functions

- void MostraVertice (Vertice *grafo)

  Função para mostar vértices as adjacencias.
- void MostraGrafo (Grafo *g)

  Função para mostra um grafo.
- char * ReadFile (char *file)

  Função para ler um arquivo.
- void Contador (char *dados, int *linha, int *coluna)

  Função para contar o número de linhas e colunas em uma string.
- Grafo * CriarVerticesCSV (char *dados)

  Função para criar vértices a partir de um arquivo CSV.
- Grafo * CarregaDadosCSV (char *file)

  Função para carregar dados de um arquivo CSV.
- void GuardaVertices (Grafo *g, char *file)

  Função para guardar vértices em um ficheiro binário.
- Grafo * CarregaVertices (char *file)

  Função para carregar vértices de um arquivo.
- void GuardarAdjacentes (Grafo *g, char *file)

  Função para guardar adjacências em um ficheiro binário.
- Grafo * CarregaAdjacencias (Grafo *grafo, char *file)

  Função para carregar adjacências de um arquivo num grafo.
- void GuardaGrafo (Grafo *g, char *vertices, char *adjacencias)

  Função para guardar um grafo em dois arquivos, um para vértices e outro para adjacências.
- Grafo * CarregaGrafo (char *vertices, char *adjacencias)

  Função para carregar um grafo a partir de dois arquivos, um para vértices e outro para adjacências.
- Grafo * CarregaDados (char *file, char *vertices, char *adjacencias)

  Função para carregar dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível.
- void ImprimirCaminho (int verticeAnt[ ], int destino)

  Mostra o caminho de um vértice de origem a um vértice de destino.
- void MostrarCaminho (Grafo *g, int origem, int destino)

  Mostra o caminho mais curto entre dois vértices num grafo.

5.19.1 Detailed Description

Este ficheiro de cabeçalho define as funções para carregar e mostrar dados a partir de ficheiros.

Author

Hugo Cruz (a23010)

Version

71.1

Date

2024-05-24

Copyright

5.19.2 Function Documentation

5.19.2.1 CarregaAdjacencias()

```
Grafo * CarregaAdjacencias (
            Grafo * grafo,
            char * file)
```

Função para carregar adjacências de um arquivo num grafo.
Esta função carrega as adjacências de um ficheiro binário para um grafo. Lê o ficheiro e adiciona cada adjacência ao grafo correspondente.

Parameters

grafo	Apontador para o grafo onde seram carregado os dados.
file	Nome do arquivo.

Returns

Grafo* Retorna um apontador para o grafo atualizado.

```
00361 {
00362     bool inf;
00363     FILE* ficheiro = fopen(file, "rb");
00364     if (ficheiro == NULL) return NULL;
00365 
00366     AdjacenteFile auxAF;
00367 
00368     //Le o cabeçalho
00369     fread(&auxAF, sizeof(auxAF), 1, ficheiro) == 1;
00370 
00371     //Se for o ficheiro com um id diferente de -8 não le
00372     if (auxAF.id != -8) return NULL;
00373 
00374     Vertice* aux = grafo->inicioGrafo;
00375 
00376 
00377     while (aux)
00378     {
00379         //Le até encontar o -1
00380         while (fread(&auxAF, sizeof(auxAF), 1, ficheiro) == 1 && auxAF.id != -1)
00381         {
00382             grafo = InserirAdjGrafo(grafo, aux->id, auxAF.id, auxAF.peso, &inf);
00383         }
00384         aux = aux->nextV;
00385     }
00386 
00387 
00388     fclose(ficheiro);
00389     return grafo;
00390 }
```

5.19.2.2 CarregaDados()

```
Grafo * CarregaDados (
            char * file,
            char * vertices,
            char * adjacencias)
```

Função para carregar dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível.
Esta função carrega dados de um arquivo ou de dois arquivos (vértices e adjacências), dependendo do que estiver disponível. Se os ficheiros de vértices e adjacências estiverem disponíveis, usa-os para carregar o grafo. Caso contrário, carrega os dados de um ficheiro CSV.

Parameters

file	Nome do arquivo com dados (argumento).
vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para vértices.

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00440 {
00441     bool inf;
00442
00443     Grafo* g = CriarGrafo(&inf);
00444
00445     g = CarregaGrafo(vertices, adjacencias);
00446
00447     //Se ocorrer algum erro com os ficheiros le o csv
00448     if (g != NULL)
00449     {
00450         return g;
00451     }
00452     else
00453     {
00454         g = CarregaDadosCSV(file);
00455         return g;
00456     }
00457
00458 }
```

5.19.2.3 CarregaDadosCSV()

```
Grafo * CarregaDadosCSV (
            char * file)
```

Função para carregar dados de um arquivo CSV.
Esta função recebe uma string de dados que representa um ficheiro CSV. Cria um grafo e insere vértices no grafo com base nos dados. O número de vértices inseridos é o maior entre o número de linhas e o número de colunas nos dados. Retorna um apontador para o grafo criado.

Parameters

file	Nome do arquivo CSV (argumento 1).

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00195 {
00196     bool inf;
00197
00198     int linhas = 0, colunas = 0;
00199     char* saveptr_linha = NULL;
00200     char* saveptr_coluna = NULL;
00201
00202     char* dados = ReadFile(file); //Le tudo do ficheiro csv
00203     char* dados_copy = strdup(dados); // Cria uma c◆pia da string lida
00204     Grafo* g = CriarVerticesCSV(dados_copy); // Conta e cria os vertices
00205
00206     //Divide uma string em tokens separados por \n
00207     dados = strtok_s(dados, "\n", &saveptr_linha);
00208
00209     while (dados != NULL)
00210     {
00211         linhas++;
00212         dados = strtok_s(dados, ";", &saveptr_coluna); //Divide uma string em tokens separados por ;
00213
00214         while (dados != NULL)
00215         {
00216             colunas++;
00217             g = InserirAdjGrafo(g, linhas, colunas, atoi(dados), &inf); //Tranforma uma string em
    inteiro
00218             dados = strtok_s(NULL, ";", &saveptr_coluna); //Coloca NULL e avança
00219         }
00220         colunas = 0;
00221         dados = strtok_s(NULL, "\n", &saveptr_linha);
00222     }
00223
00224     free(dados);
00225     return g;
00226
00227 }
```

5.19.2.4 CarregaGrafo()

```
Grafo * CarregaGrafo (
            char * vertices,
            char * adjacencias)
```

Função para carregar um grafo a partir de dois arquivos, um para vértices e outro para adjacências.
Esta função carrega um grafo inteiro a partir de dois ficheiros, um para os vértices e outro para as adjacências.
Usa as funções CarregaVertices e CarregaAdjacencias

Parameters

vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para adjacências.

Returns

Grafo* Retorna um apontador para o grafo com os valores carregados

```
00419 {
00420     Grafo* g = CarregaVertices(vertices);
00421     if (g == NULL)return NULL;
00422     Grafo* grafo = CarregaAdjacencias(g, adjacencias);
00423
00424     return grafo;
00425 }
```

5.19.2.5 CarregaVertices()

```
Grafo * CarregaVertices (
            char * file)
```

Função para carregar vértices de um arquivo.
Esta função carrega os vértices de um ficheiro binário para um grafo. Lê o ficheiro e adiciona cada vértice ao grafo.

Parameters

file	Nome do arquivo

Returns

Grafo* Retorna um apontador para o grafo criado

```
00272 {
00273     bool inf;
00274     FILE* ficheiro = fopen(file, "rb");
00275
00276     if (ficheiro == NULL) return;
00277
00278     Grafo* g = CriarGrafo(&inf);//Cria um grafo
00279
00280     VerticeFile auxVF;
00281     Vertice* novo = NULL;
00282
00283     //Le o cabeçalho
00284     fread(&auxVF, sizeof(auxVF), 1, ficheiro) == 1;
00285
00286     //Se for o ficheiro correto contínua a ler
00287     if (auxVF.id == -7)
00288     {
00289         while (fread(&auxVF, sizeof(auxVF), 1, ficheiro) == 1)
00290         {
00291             g = InserirVerticeGrafo(g, auxVF.id, &inf);
00292         }
00293     }
00294     else
00295     {
00296         return NULL;
00297     }
00298
00299     fclose(ficheiro);//Fecha o ficheiro
00300     return g;
00301 }
```

5.19.2.6 Contador()

```
void Contador (
            char * dados,
            int * linha,
            int * coluna)
```

Função para contar o número de linhas e colunas em uma string.
Esta função recebe uma string de dados e dois apontadores para inteiros. Conta o número de linhas e colunas na string de dados e armazena esses valores nos inteiros apontados pelos apontadores.

Parameters

dados	Dados a serem contados.
linha	Apontador para o número de linhas.
coluna	Apontador para o número de colunas.

```
00103 {
00104     char* saveptr_linha = NULL;
00105     char* saveptr_coluna = NULL;
00106     int colunasNaLinha = 0;
00107     *linha = 0;
00108     *coluna = 0;
00109
00110     //Divide uma string em tokens separados por \n
00111     dados = strtok_s(dados, "\n", &saveptr_linha);
00112
00113     while (dados != NULL)
00114     {
00115         (*linha)++;
00116
00117         //Divide uma string em tokens separados por ;
00118         dados = strtok_s(dados, ";", &saveptr_coluna);
00119
00120         while (dados != NULL)
00121         {
00122             colunasNaLinha++;
00123             //Coloca NULL e avança
00124             dados = strtok_s(NULL, ";", &saveptr_coluna);
00125         }
00126
00127         //Guarda o número maior de linhas contadas
00128         if (colunasNaLinha > *coluna)
00129         {
00130             *coluna = colunasNaLinha;
00131         }
00132
00133         colunasNaLinha = 0;
00134
00135         dados = strtok_s(NULL, "\n", &saveptr_linha);
00136     }
00137
00138 }
```

5.19.2.7 CriarVerticesCSV()

```
Grafo * CriarVerticesCSV (
            char * dados)
```

Função para criar vértices a partir de um arquivo CSV.
Esta função conta as linhas e colunas de um token e cria memória para um grafo. Após a criação do mesmo carrega os vertices de um ficheiro CSV.

Parameters

dados	Dados do arquivo CSV.

Returns

Grafo* Retorna um apontador para o grafo criado.

```
00150 {
00151     bool inf;
00152     int linhas = 0;
00153     int colunas = 0;
00154 
00155     //Conta linhas e colunas
00156     Contador(dados, &linhas, &colunas);
00157 
00158     Grafo* g = CriarGrafo(&inf);
00159 
00160     if (linhas == NULL || colunas == NULL) return NULL;
00161     //Cria o número de vertices que corresponde ao maior valor
00162     if (linhas <= colunas)
00163     {
00164         for (int i = 1; i <= colunas; i++)
00165         {
00166             g = InserirVerticeGrafo(g, i, &inf);
00167 
00168         }
00169     }
00170     else
00171     {
00172         for (int i = 1; i <= linhas; i++)
00173         {
00174             g = InserirVerticeGrafo(g, i, &inf);
00175 
00176         }
00177     }
00178 
00179     free(dados);
00180     return g;
00181 
00182 }
```

5.19.2.8 GuardaGrafo()

```
void GuardaGrafo (
            Grafo * g,
            char * vertices,
            char * adjacencias)
```

Função para guardar um grafo em dois arquivos, um para vértices e outro para adjacências.
Esta função guarda um grafo inteiro em dois ficheiros, um para os vértices e outro para as adjacências. Usa as funções GuardaVertices e GuardarAdjacentes

Parameters

g	Apontador para o grafo a guardar.
vertices	Nome do arquivo para vértices.
adjacencias	Nome do arquivo para adjacências.

```
00403 {
00404     GuardaVertices(g, vertices);
00405     GuardarAdjacentes(g, adjacencias);
00406 }
```

5.19.2.9 GuardarAdjacentes()

```
void GuardarAdjacentes (
            Grafo * g,
            char * file)
```

Função para guardar adjacências em um ficheiro binário.
Esta função guarda as adjacências de um grafo num ficheiro binário. Percorre todos os vértices e as suas adjacências e escreve-os no ficheiro.

Parameters

g	Apontador para o grafo a guardar
file	Nome do arquivo

```
00313 {
00314     FILE* ficheiro = fopen(file, "wb");
00315
00316     if (ficheiro == NULL) return;
00317
00318     Vertice* auxV = g->inicioGrafo;
00319     AdjacenteFile auxAF; // Estrutura sem apontadores
00320
00321     //Adicina um cabeçalho para garantir que le o ficheiro correto
00322     auxAF.id = -8;
00323     fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00324
00325     //Avança com os vertices, mas apenas escreve as adjacências
00326     while (auxV)
00327     {
00328         Adjacente* auxA = auxV->nextA;
00329
00330         while (auxA)
00331         {
00332             auxAF.id = auxA->id;
00333             auxAF.peso = auxA->peso;
00334
00335             fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00336
00337             auxA = auxA->next;
00338         }
00339
00340         //Marca para sair do while na leitura
00341         auxAF.id = -1;
00342         fwrite(&auxAF, sizeof(AdjacenteFile), 1, ficheiro);
00343
00344         auxV = auxV->nextV;
00345     }
00346
00347     fclose(ficheiro);
00348 }
```

5.19.2.10 GuardaVertices()

```
void GuardaVertices (
            Grafo * g,
            char * file)
```

Função para guardar vértices em um ficheiro binário.
Esta função guarda os vértices de um grafo num ficheiro binário. Percorre todos os vértices do grafo e escreve-os no ficheiro.

Parameters

g	Apontador para o grafo a guardar.
file	Nome do arquivo.

```
00239 {
00240     FILE* ficheiro = fopen(file, "wb");
00241
00242     if (ficheiro == NULL) return;
00243
00244     Vertice* auxV = g->inicioGrafo;
00245     VerticeFile auxVF; // Estrutura sem apontadores
00246
00247     //Adicina um cabeçalho para garantir que le o ficheiro correto
00248     auxVF.id = -7;
00249     fwrite(&auxVF, sizeof(VerticeFile), 1, ficheiro);
00250
00251     //Escreve todos os vertices em modo binário
00252     while (auxV)
00253     {
00254         auxVF.id = auxV->id;
00255         fwrite(&auxVF, sizeof(VerticeFile), 1, ficheiro);
00256         auxV = auxV->nextV;
00257     }
00258
00259     fclose(ficheiro);  // Fecha o ficheiro após a escrita
00260 }
```

5.19.2.11 ImprimirCaminho()

```
void ImprimirCaminho (
            int verticeAnt[],
            int destino)
```

Mostra o caminho de um vértice de origem a um vértice de destino.

Parameters

verticeAnt	Array que contém os antecessore de cada vertice
destino	Destino final

```
00467 {
00468     int caminho[MAX];
00469     int contador = 0;
00470     int atual = destino;
00471 
00472      // Coloca num arrays todos os anteriores a cada vertice
00473     while (atual != -1)
00474     {
00475         caminho[contador++] = atual;
00476         atual = verticeAnt[atual];
00477     }
00478 
00479     // Imprime o caminho do vértice de origem ao vértice de destino
00480     for (int i = contador - 1; i >= 0; i--)
00481     {
00482         printf("%d ", caminho[i]);
00483     }
00484 }
```

5.19.2.12 MostraGrafo()

```
void MostraGrafo (
            Grafo * g)
```

Função para mostra um grafo.
Esta função recebe um apontador para um grafo e imprime o grafo chamando a função MostraVertice para o vértice inicial do grafo.

Parameters

g	Apontador para o grafo a ser mostrado

```
00054 {
00055     MostraVertice(g->inicioGrafo);
00056 }
```

5.19.2.13 MostrarCaminho()

```
void MostrarCaminho (
            Grafo * g,
            int origem,
            int destino)
```

Mostra o caminho mais curto entre dois vértices num grafo.
Esta função recebe um grafo, um vértice de origem e um vértice de destino. Ela utiliza o algoritmo de Dijkstra para calcular o caminho mais curto do vértice de origem ao vértice de destino e imprime esse caminho.

Parameters

g	O grafo.
origem	O vértice de origem.
destino	O vértice de destino.

```
00497 {
00498     if (g == NULL)
00499     {
```

```
00500         return;
00501     }
00502
00503     int distanciasFinais[MAX];
00504     int verticeAnt[MAX];
00505
00506     // Inicializa o array de vértices anteriores
00507     for (int i = 0; i < MAX; i++)
00508     {
00509         verticeAnt[i] = -1;
00510     }
00511
00512     // Inicializa o array de vértices anteriores
00513     InicializarArrays(distanciasFinais);
00514
00515     Dijkstra(g, origem, distanciasFinais, verticeAnt);
00516
00517     if (distanciasFinais[destino] == INT_MAX)
00518     {
00519         printf("Não existe caminho de %d para %d\n", origem, destino);
00520     }
00521     else
00522     {
00523         printf("Caminho de %d para %d: ", origem, destino);
00524         ImprimirCaminho(verticeAnt, destino);
00525         printf("\nDistância: %d\n", distanciasFinais[destino]);
00526     }
00527 }
```

5.19.2.14 MostraVertice()

```
void MostraVertice (
            Vertice * grafo)
```

Função para mostar vértices as adjacencias.
Esta função recebe um apontador para um vértice e imprime o vértice e as suas adjacências. Percorre a lista de vértices e para cada vértice, percorre a lista de adjacências, imprimindo as.

Parameters

grafo	Apontador para o vértice a ser mostrado.

Função para mostar vértices as adjacencias.
Esta função recebe um apontador para um vértice e imprime o vértice e as suas adjacências. Percorre a lista de vértices e para cada vértice, percorre a lista de adjacências, imprimindo as.

Parameters

grafo	Apontador para o vértice a ser mostrado.

```
00021 {
00022     Vertice* aux = grafo;
00023
00024     //Avança na lista de vertices e arestas e mostra todos os pesos das adjacência
00025     while (aux != NULL)
00026     {
00027         printf("\nVértice: %d\n\n", aux->id);
00028
00029         Adjacente* adj = aux->nextA;
00030
00031         while (adj)
00032         {
00033             if (adj->peso > 0 && adj->peso != INT_MAX)
00034             {
00035                 printf("\tAdj: %d - (%d)\n", adj->id, adj->peso);
00036
00037             }
00038             adj = adj->next;
00039         }
00040
00041         aux = aux->nextV;
00042     }
00043     printf("\n");
00044 }
```

5.19.2.15 ReadFile()

```
char * ReadFile (
            char * file)
```

Função para ler um arquivo.
Esta função recebe o nome de um ficheiro, lê o ficheiro e retorna os dados lidos. Se o ficheiro não puder ser aberto, retorna NULL. Caso contrário, lê o ficheiro inteiro para uma string e retorna essa string.

Parameters

file	Nome do arquivo a ser lido.

Returns

char* Retorna os dados lidos do arquivo.

```
00067 {
00068     FILE* ficheiro = fopen(file, "r");
00069     if (ficheiro == NULL) return NULL;
00070 
00071     fseek(ficheiro, 0, SEEK_END); //Coloca o apontador para fille no fim do arquivo
00072     int tamanho = ftell(ficheiro); //Calcula o tamanho em bytes
00073     fseek(ficheiro, 0, SEEK_SET); //Coloca o apontador para fille no início do arquivo
00074 
00075     char* dados = (char*)malloc(sizeof(char) * tamanho + 1); //Aloca memória para o tamanho do
      ficheiro + 1
00076 
00077     //Informa me o tamanho lido em bytes coloca o final da string como NULL
00078     if (dados != NULL)
00079     {
00080         size_t bytesRead = fread(dados, 1, tamanho, ficheiro);
00081         dados[bytesRead] = '\0';
00082     }
00083     else
00084     {
00085         free(dados);
00086         dados = NULL;
00087     }
00088 
00089     fclose(ficheiro);
00090     return dados;
00091 }
```

5.20 InputOutput.h

Go to the documentation of this file.

```
00001 
00011 #ifndef INPUTOUTPUT_H
00012 #define INPUTOUTPUT_H
00013 
00014 #include <stdio.h>
00015 #include <string.h>
00016 #include <locale.h>
00017 #include "grafo.h"
00018 
00019 
00027 void MostraVertice(Vertice* grafo);
00028 
00036 void MostraGrafo(Grafo* g);
00037 
00038 
00047 char* ReadFile(char* file);
00048 
00058 void Contador(char* dados, int* linha, int* coluna);
00059 
00069 Grafo* CriarVerticesCSV(char* dados);
00070 
00081 Grafo* CarregaDadosCSV(char* file);
00082 
00092 void GuardaVertices(Grafo* g, char* file);
00093 
00103 Grafo* CarregaVertices(char* file);
00104 
00105 
00115 void GuardarAdjacentes(Grafo* g, char* file);
00126 Grafo* CarregaAdjacencias(Grafo* grafo, char* file);
00127 
```

```
00128
00139 void GuardaGrafo(Grafo* g, char* vertices, char* adjacencias);
00150 Grafo* CarregaGrafo(char* vertices, char* adjacencias);
00163 Grafo* CarregaDados(char* file, char* vertices, char* adjacencias);
00164
00171 void ImprimirCaminho(int verticeAnt[], int destino);
00172
00183 void MostrarCaminho(Grafo* g, int origem, int destino);
00184
00185
00186
00187
00188 #endif
```

5.21 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/vertices.c File Reference

Implementação de funções para manipular vértices.
`#include "vertices.h"`

Functions

- void ApagarVertice (Vertice *v)

  Liberta a memória alocada para um vértice.
- Vertice * CriarVertice (int id)

  Aloca memória para um novo vértice.
- bool ExisteVertice (Vertice *inicio, int id)

  Verifica a existência de um vértice com um determinado identificador.
- Vertice * InserirVerticeLista (Vertice *inicio, int id, bool *inf)

  Insere um novo vértice na lista de vértices.
- Vertice * ColocaNumaPosicaoLista (Vertice *inicio, int id, bool *inf)

  Posiciona a lista no vértice com o identificador fornecido.
- Vertice * EliminarVertice (Vertice *inicio, int id, bool *inf)

  Elimina um vértice da lista de vértices.
- Vertice * EliminarTodasAdjacenciasVertice (Vertice *inicio, int id, bool *inf)

  Elimina todas as adjacências de um vértice.

5.21.1 Detailed Description

Implementação de funções para manipular vértices.

Author

Hugo Cruz (a23010)

Version

170.0

Date

2024-05-23

Copyright

5.21.2 Function Documentation

5.21.2.1 ApagarVertice()

```
void ApagarVertice (
            Vertice * v)
```

Liberta a memória alocada para um vértice.
Esta função liberta memória alocada para um vértices

Parameters

v	Apontador para o vértice (Vertice) a ser liberado.

```
00021 {
00022     free(v);
00023 }
```

5.21.2.2 ColocaNumaPosicaoLista()

```
Vertice * ColocaNumaPosicaoLista (
            Vertice * inicio,
            int id,
            bool * inf)
```

Posiciona a lista no vértice com o identificador fornecido.
Esta função posiciona a lista no vértice com o identificador fornecido. Retorna um apontador para o vértice encontrado ou NULL se não for encontrado e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser encontrado.
inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o vértice encontrado ou NULL se não for encontrado.

```
00159 {
00160     *inf = false;
00161
00162     if (inicio == NULL)
00163     {
00164         *inf = false;
00165         return NULL;
00166     }
00167
00168     Vertice* aux = inicio;
00169
00170     //Encontra uma posição e retorna a posição atual e não o inicio da lista
00171     while (aux)
00172     {
00173         if (aux->id == id)
00174         {
00175             *inf = true;
00176             return aux;
00177         }
00178         aux = aux->nextV;
00179     }
00180
00181     *inf = false;
00182     return NULL;
00183 }
```

5.21.2.3 CriarVertice()

```
Vertice * CriarVertice (
            int id)
```

Aloca memória para um novo vértice.

Esta função aloca memória para um novo vértice. Retorna um apontador para o vértice criado e atualiza o valor de inf para indicar o sucesso ou falha na criação do vértice.

Parameters

id	Aloca memória para um novo vértice.

Returns

Apontador para o vértice criado.

```
00035 {
00036     Vertice* aux = (Vertice*)malloc(sizeof(Vertice)); //Aloca memória para uma estrutura vertice
00037
00038     if (aux == NULL) return NULL;
00039
00040     aux->id = id;
00041     aux->nextV = NULL;
00042     aux->nextA = NULL;
00043
00044     return aux;
00045 }
```

5.21.2.4 EliminarTodasAdjacenciasVertice()

```
Vertice * EliminarTodasAdjacenciasVertice (
            Vertice * inicio,
            int id,
            bool * inf)
```

Elimina todas as adjacências de um vértice.
sta função elimina todas as adjacências de um vértice. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice cujas adjacências serão eliminadas.
inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o início da lista de vértices.

```
00252 {
00253     *inf = false;
00254
00255     if (inicio == NULL)
00256     {
00257         return NULL;
00258     }
00259
00260     // Percorre todos os vértices da lista
00261     Vertice* aux = inicio;
00262
00263     while (aux)
00264     {
00265         // Remove a adjacência do vértice atual
00266         aux->nextA = EliminaUmaAdj(aux->nextA, id, &inf);
00267         aux = aux->nextV;
00268     }
00269
00270     *inf = true;
00271
00272     return inicio;
00273 }
```

5.21.2.5 EliminarVertice()

```
Vertice * EliminarVertice (
            Vertice * inicio,
            int id,
            bool * inf)
```

Elimina um vértice da lista de vértices.
Esta função elimina um vértice da lista de vértices. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser eliminado.
inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o início da lista de vértices.

```
00197 {
00198     *inf = false;
00199
00200     if (inicio == NULL)
00201     {
00202         return NULL;
00203     }
00204
00205     //Ajuda a encontar a posição para inserir o vértice
00206     Vertice* ant = NULL;
00207     Vertice* aux = inicio;
00208
00209
00210     while (aux->id != id)
00211     {
00212         ant = aux;
00213         aux = aux->nextV;
00214     }
00215
00216     if (!aux)
00217     {
00218         return inicio;
00219     }
00220
00221     // Se o vértice a ser removido é o primeiro da lista
00222     if (ant == NULL)
00223     {
00224         aux->nextA = ElimanaTodasAdj(aux->nextA, &inf);
00225         if (*inf == false) return inicio;
00226         inicio = aux->nextV;
00227     }
00228     else
00229     {
00230         // Caso contrário, remove o vértice da lista
00231         ant->nextV = aux->nextV;
00232     }
00233
00234     // Elimina todas as adjacências do vértice e apaga o vértice
00235     ApagarVertice(aux);
00236     *inf = true;
00237     return inicio;
00238 }
```

5.21.2.6 ExisteVertice()

```
bool ExisteVertice (
            Vertice * inicio,
            int id)
```

Verifica a existência de um vértice com um determinado identificador.
Esta função verifica a existência de um vértice com um determinado identificador na lista de vértices. Retorna verdadeiro se o vértice existir, falso caso contrário.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser verificado.

Returns

Retorna verdadeiro se o vértice existir, falso caso contrário.

```
00057 {
00058     Vertice* aux = inicio;
00059 
00060     //Corre a lista até encontar o vertice procurado
00061     while (aux)
00062     {
00063         if (aux->id == id)
00064         {
00065             return true;
00066         }
00067 
00068         aux = aux->nextV;
00069     }
00070 
00071     return false;
00072 
00073 }
```

5.21.2.7 InserirVerticeLista()

```
Vertice * InserirVerticeLista (
            Vertice * inicio,
            int id,
            bool * inf)
```

Insere um novo vértice na lista de vértices.
Esta função cria e insere um novo vértice na lista de vértices. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na inserção do vértice.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do novo vértice.
inf	Apontador para um bool que indica se a execução correu bem.

Returns

Apontador para um booleano que indica o sucesso ou falha na inserção do vértice.

```
00088 {
00089     *inf = false;
00090 
00091     Vertice* novo = CriarVertice(id);
00092 
00093     *inf = false;
00094 
00095 
00096     if (novo == NULL)
00097     {
00098         return inicio;
00099     }
00100 
00101     //Coloca o novo vertice no inicio da lista
00102     if (inicio == NULL)
00103     {
00104         inicio = novo;
00105         *inf = true;
00106         return inicio;
00107     }
00108 
00109     // Se o vertice já existir não cria elimina o vertice criado
00110     if (ExisteVertice(inicio, novo->id))
00111     {
00112         free(novo);
00113         return inicio;
00114     }
00115     else
00116     {
00117         //Ajuda a encontar a posição para inserir o vértice
00118         Vertice* ant = NULL;
00119         Vertice* aux = inicio;
00120 
00121         // Percorre a lista até encontrar um vértice com um id maior que o novo vértice
```

```
00122         while (aux != NULL && aux->id < novo->id)
00123         {
00124             ant = aux;
00125             aux = aux->nextV;
00126         }
00127 
00128         // Se o ant for null insere no início da lista
00129         if (ant == NULL)
00130         {
00131             novo->nextV = inicio;
00132             inicio = novo;
00133         }
00134         else
00135         {
00136             // Caso contrário, insere o novo vértice na posição correta
00137             novo->nextV = aux;
00138             ant->nextV = novo;
00139         }
00140 
00141         *inf = true;
00142         return inicio;
00143     }
00144 }
```

5.22 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Grafos/vertices.h File Reference

Este ficheiro contém as definições das estruturas de dados para os vértices num grafo.

```
#include "adjacente.h"
```

Data Structures

- struct Vertice

  Estrutura de um vértice num grafo.
- struct VerticeFile

  Estrutura de um vértice utilizada para armazenar vértices em um ficheiro binário.

Typedefs

- typedef struct Vertice Vertice

  Estrutura de um vértice num grafo.
- typedef struct VerticeFile **VerticeFile**

  Estrutura de um vértice utilizada para armazenar vértices em um ficheiro binário.

Functions

- void ApagarVertice (Vertice *v)

  Liberta a memória alocada para um vértice.
- Vertice * CriarVertice (int id)

  Aloca memória para um novo vértice.
- bool ExisteVertice (Vertice *inicio, int id)

  Verifica a existência de um vértice com um determinado identificador.
- Vertice * InserirVerticeLista (Vertice *inicio, int id, bool *inf)

  Insere um novo vértice na lista de vértices.
- Vertice * ColocaNumaPosicaoLista (Vertice *inicio, int id, bool *inf)

  Posiciona a lista no vértice com o identificador fornecido.
- Vertice * EliminarVertice (Vertice *inicio, int id, bool *inf)

  Elimina um vértice da lista de vértices.
- Vertice * EliminarTodasAdjacenciasVertice (Vertice *inicio, int id, bool *inf)

  Elimina todas as adjacências de um vértice.

5.22.1 Detailed Description

Este ficheiro contém as definições das estruturas de dados para os vértices num grafo.

Author

Hugo Cruz (a23010)

Version

92.1

Date

2024-05-24

Copyright

5.22.2 Typedef Documentation

5.22.2.1 Vertice

```
typedef struct Vertice Vertice
```

Estrutura de um vértice num grafo.
Esta estrutura representa um vértice num grafo. Cada vértice tem um identificador único (id), um apontador para o próximo vértice na lista de vértices (nextV), um apontador para o primeiro adjacente na lista de adjacências (nextA) e um indicador se o vértice foi visitado ou não (visitado).

5.22.3 Function Documentation

5.22.3.1 ApagarVertice()

```
void ApagarVertice (
            Vertice * v)
```

Liberta a memória alocada para um vértice.
Esta função liberta memória alocada para um vértices

Parameters

v	Apontador para o vértice (Vertice) a ser liberado.

```
00021 {
00022     free(v);
00023 }
```

5.22.3.2 ColocaNumaPosicaoLista()

```
Vertice * ColocaNumaPosicaoLista (
            Vertice * inicio,
            int id,
            bool * inf)
```

Posiciona a lista no vértice com o identificador fornecido.
Esta função posiciona a lista no vértice com o identificador fornecido. Retorna um apontador para o vértice encontrado ou NULL se não for encontrado e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser encontrado.
inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o vértice encontrado ou NULL se não for encontrado.

```
00159 {
00160     *inf = false;
00161
00162     if (inicio == NULL)
00163     {
00164         *inf = false;
00165         return NULL;
00166     }
00167
00168     Vertice* aux = inicio;
00169
00170     //Encontra uma posição e retorna a posição atual e não o inicio da lista
00171     while (aux)
00172     {
00173         if (aux->id == id)
00174         {
00175             *inf = true;
00176             return aux;
00177         }
00178         aux = aux->nextV;
00179     }
00180
00181     *inf = false;
00182     return NULL;
00183 }
```

5.22.3.3 CriarVertice()

```
Vertice * CriarVertice (
            int id)
```

Aloca memória para um novo vértice.
Esta função aloca memória para um novo vértice. Retorna um apontador para o vértice criado e atualiza o valor de inf para indicar o sucesso ou falha na criação do vértice.

Parameters

id	Aloca memória para um novo vértice.

Returns

Apontador para o vértice criado.

```
00035 {
00036     Vertice* aux = (Vertice*)malloc(sizeof(Vertice)); //Aloca memória para uma estrutura vertice
00037
00038     if (aux == NULL) return NULL;
00039
00040     aux->id = id;
00041     aux->nextV = NULL;
00042     aux->nextA = NULL;
00043
00044     return aux;
00045 }
```

5.22.3.4 EliminarTodasAdjacenciasVertice()

```
Vertice * EliminarTodasAdjacenciasVertice (
            Vertice * inicio,
            int id,
            bool * inf)
```

Elimina todas as adjacências de um vértice.
sta função elimina todas as adjacências de um vértice. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice cujas adjacências serão eliminadas.

inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o início da lista de vértices.

```
00252 {
00253     *inf = false;
00254
00255     if (inicio == NULL)
00256     {
00257         return NULL;
00258     }
00259
00260     // Percorre todos os vértices da lista
00261     Vertice* aux = inicio;
00262
00263     while (aux)
00264     {
00265         // Remove a adjacência do vértice atual
00266         aux->nextA = EliminaUmaAdj(aux->nextA, id, &inf);
00267         aux = aux->nextV;
00268     }
00269
00270     *inf = true;
00271
00272     return inicio;
00273 }
```

5.22.3.5 EliminarVertice()

```
Vertice * EliminarVertice (
            Vertice * inicio,
            int id,
            bool * inf)
```

Elimina um vértice da lista de vértices.
Esta função elimina um vértice da lista de vértices. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser eliminado.
inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o início da lista de vértices.

```
00197 {
00198     *inf = false;
00199
00200     if (inicio == NULL)
00201     {
00202         return NULL;
00203     }
00204
00205     //Ajuda a encontar a posição para inserir o vértice
00206     Vertice* ant = NULL;
00207     Vertice* aux = inicio;
00208
00209
00210     while (aux->id != id)
00211     {
00212         ant = aux;
00213         aux = aux->nextV;
00214     }
00215
00216     if (!aux)
00217     {
00218         return inicio;
00219     }
```

```
00220
00221     // Se o vértice a ser removido é o primeiro da lista
00222     if (ant == NULL)
00223     {
00224         aux->nextA = ElimanaTodasAdj(aux->nextA, &inf);
00225         if (*inf == false) return inicio;
00226         inicio = aux->nextV;
00227     }
00228     else
00229     {
00230         // Caso contrário, remove o vértice da lista
00231         ant->nextV = aux->nextV;
00232     }
00233
00234     // Elimina todas as adjacências do vértice e apaga o vértice
00235     ApagarVertice(aux);
00236     *inf = true;
00237     return inicio;
00238 }
```

5.22.3.6 ExisteVertice()

```
bool ExisteVertice (
            Vertice * inicio,
            int id)
```

Verifica a existência de um vértice com um determinado identificador.
Esta função verifica a existência de um vértice com um determinado identificador na lista de vértices. Retorna verdadeiro se o vértice existir, falso caso contrário.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser verificado.

Returns

Retorna verdadeiro se o vértice existir, falso caso contrário.

```
00057 {
00058     Vertice* aux = inicio;
00059
00060     //Corre a lista até encontar o vertice procurado
00061     while (aux)
00062     {
00063         if (aux->id == id)
00064         {
00065             return true;
00066         }
00067
00068         aux = aux->nextV;
00069     }
00070
00071     return false;
00072
00073 }
```

5.22.3.7 InserirVerticeLista()

```
Vertice * InserirVerticeLista (
            Vertice * inicio,
            int id,
            bool * inf)
```

Insere um novo vértice na lista de vértices.
Esta função cria e insere um novo vértice na lista de vértices. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na inserção do vértice.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do novo vértice.

inf	Apontador para um bool que indica se a execução correu bem.

Returns

Apontador para um booleano que indica o sucesso ou falha na inserção do vértice.

```
00088 {
00089     *inf = false;
00090 
00091     Vertice* novo = CriarVertice(id);
00092 
00093     *inf = false;
00094 
00095 
00096     if (novo == NULL)
00097     {
00098         return inicio;
00099     }
00100 
00101     //Coloca o novo vertice no inicio da lista
00102     if (inicio == NULL)
00103     {
00104         inicio = novo;
00105         *inf = true;
00106         return inicio;
00107     }
00108 
00109     // Se o vertice já existir não cria elimina o vertice criado
00110     if (ExisteVertice(inicio, novo->id))
00111     {
00112         free(novo);
00113         return inicio;
00114     }
00115     else
00116     {
00117         //Ajuda a encontar a posição para inserir o vértice
00118         Vertice* ant = NULL;
00119         Vertice* aux = inicio;
00120 
00121         // Percorre a lista até encontrar um vértice com um id maior que o novo vértice
00122         while (aux != NULL && aux->id < novo->id)
00123         {
00124             ant = aux;
00125             aux = aux->nextV;
00126         }
00127 
00128         // Se o ant for null insere no início da lista
00129         if (ant == NULL)
00130         {
00131             novo->nextV = inicio;
00132             inicio = novo;
00133         }
00134         else
00135         {
00136             // Caso contrário, insere o novo vértice na posição correta
00137             novo->nextV = aux;
00138             ant->nextV = novo;
00139         }
00140 
00141         *inf = true;
00142         return inicio;
00143     }
00144 }
```

5.23 vertices.h

Go to the documentation of this file.

```
00001 
00012 #ifndef VERTICE_H
00013 #define VERTICE_H
00014 
00015 #include "adjacente.h"
00016 
00024 typedef struct Vertice
00025 {
00026     int id;
00027     struct Vertice *nextV;
00028     Adjacente *nextA;
00029 
00030 } Vertice;
```

```
00031
00036 typedef struct VerticeFile
00037 {
00038     int id;
00039
00040 } VerticeFile;
00041
00049 void ApagarVertice(Vertice *v);
00050
00060 Vertice* CriarVertice(int id);
00061
00071 bool ExisteVertice(Vertice *inicio, int id);
00072
00085 Vertice *InserirVerticeLista(Vertice *inicio, int id, bool *inf);
00086
00099 Vertice *ColocaNumaPosicaoLista(Vertice *inicio, int id, bool *inf);
00100
00112 Vertice *EliminarVertice(Vertice *inicio, int id, bool *inf);
00113
00125 Vertice *EliminarTodasAdjacenciasVertice(Vertice *inicio, int id, bool *inf);
00126
00127 #endif
```

5.24 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Main/libs/vertices.h File Reference

Este ficheiro contém as definições das estruturas de dados para os vértices num grafo.
`#include "adjacente.h"`

Data Structures

- struct Vertice

  Estrutura de um vértice num grafo.
- struct VerticeFile

  Estrutura de um vértice utilizada para armazenar vértices em um ficheiro binário.

Typedefs

- typedef struct Vertice Vertice

  Estrutura de um vértice num grafo.
- typedef struct VerticeFile **VerticeFile**

  Estrutura de um vértice utilizada para armazenar vértices em um ficheiro binário.

Functions

- void ApagarVertice (Vertice *v)

  Liberta a memória alocada para um vértice.
- Vertice * CriarVertice (int id)

  Aloca memória para um novo vértice.
- bool ExisteVertice (Vertice *inicio, int id)

  Verifica a existência de um vértice com um determinado identificador.
- Vertice * InserirVerticeLista (Vertice *inicio, int id, bool *inf)

  Insere um novo vértice na lista de vértices.
- Vertice * ColocaNumaPosicaoLista (Vertice *inicio, int id, bool *inf)

  Posiciona a lista no vértice com o identificador fornecido.
- Vertice * EliminarVertice (Vertice *inicio, int id, bool *inf)

  Elimina um vértice da lista de vértices.
- Vertice * EliminarTodasAdjacenciasVertice (Vertice *inicio, int id, bool *inf)

  Elimina todas as adjacências de um vértice.

5.24.1 Detailed Description

Este ficheiro contém as definições das estruturas de dados para os vértices num grafo.

Author

Hugo Cruz (a23010)

Version

92.1

Date

2024-05-24

Copyright

Copyright (c) 2024

5.24.2 Typedef Documentation

5.24.2.1 Vertice

```
typedef struct Vertice Vertice
```

Estrutura de um vértice num grafo.
Esta estrutura representa um vértice num grafo. Cada vértice tem um identificador único (id), um apontador para o próximo vértice na lista de vértices (nextV), um apontador para o primeiro adjacente na lista de adjacências (nextA) e um indicador se o vértice foi visitado ou não (visitado).

5.24.3 Function Documentation

5.24.3.1 ApagarVertice()

```
void ApagarVertice (
            Vertice * v)
```

Liberta a memória alocada para um vértice.
Esta função liberta memória alocada para um vértices

Parameters

v	Apontador para o vértice (Vertice) a ser liberado.

```
00021 {
00022     free(v);
00023 }
```

5.24.3.2 ColocaNumaPosicaoLista()

```
Vertice * ColocaNumaPosicaoLista (
            Vertice * inicio,
            int id,
            bool * inf)
```

Posiciona a lista no vértice com o identificador fornecido.
Esta função posiciona a lista no vértice com o identificador fornecido. Retorna um apontador para o vértice encontrado ou NULL se não for encontrado e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser encontrado.
inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o vértice encontrado ou NULL se não for encontrado.

```
00159 {
00160     *inf = false;
00161
00162     if (inicio == NULL)
00163     {
00164         *inf = false;
00165         return NULL;
00166     }
00167
00168     Vertice* aux = inicio;
00169
00170     //Encontra uma posição e retorna a posição atual e não o inicio da lista
00171     while (aux)
00172     {
00173         if (aux->id == id)
00174         {
00175             *inf = true;
00176             return aux;
00177         }
00178         aux = aux->nextV;
00179     }
00180
00181     *inf = false;
00182     return NULL;
00183 }
```

5.24.3.3 CriarVertice()

```
Vertice * CriarVertice (
            int id)
```

Aloca memória para um novo vértice.
Esta função aloca memória para um novo vértice. Retorna um apontador para o vértice criado e atualiza o valor de inf para indicar o sucesso ou falha na criação do vértice.

Parameters

id	Aloca memória para um novo vértice.

Returns

Apontador para o vértice criado.

```
00035 {
00036     Vertice* aux = (Vertice*)malloc(sizeof(Vertice)); //Aloca memória para uma estrutura vertice
00037
00038     if (aux == NULL) return NULL;
00039
00040     aux->id = id;
00041     aux->nextV = NULL;
00042     aux->nextA = NULL;
00043
00044     return aux;
00045 }
```

5.24.3.4 EliminarTodasAdjacenciasVertice()

```
Vertice * EliminarTodasAdjacenciasVertice (
            Vertice * inicio,
            int id,
            bool * inf)
```

Elimina todas as adjacências de um vértice.
sta função elimina todas as adjacências de um vértice. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice cujas adjacências serão eliminadas.

inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o início da lista de vértices.

```
00252 {
00253     *inf = false;
00254
00255     if (inicio == NULL)
00256     {
00257         return NULL;
00258     }
00259
00260     // Percorre todos os vértices da lista
00261     Vertice* aux = inicio;
00262
00263     while (aux)
00264     {
00265         // Remove a adjacência do vértice atual
00266         aux->nextA = EliminaUmaAdj(aux->nextA, id, &inf);
00267         aux = aux->nextV;
00268     }
00269
00270     *inf = true;
00271
00272     return inicio;
00273 }
```

5.24.3.5 EliminarVertice()

```
Vertice * EliminarVertice (
            Vertice * inicio,
            int id,
            bool * inf)
```

Elimina um vértice da lista de vértices.
Esta função elimina um vértice da lista de vértices. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na operação.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser eliminado.
inf	Apontador para um booleano que indica o sucesso ou falha na operação.

Returns

Apontador para o início da lista de vértices.

```
00197 {
00198     *inf = false;
00199
00200     if (inicio == NULL)
00201     {
00202         return NULL;
00203     }
00204
00205     //Ajuda a encontar a posição para inserir o vértice
00206     Vertice* ant = NULL;
00207     Vertice* aux = inicio;
00208
00209
00210     while (aux->id != id)
00211     {
00212         ant = aux;
00213         aux = aux->nextV;
00214     }
00215
00216     if (!aux)
00217     {
00218         return inicio;
00219     }
```

```
00220
00221     // Se o vértice a ser removido é o primeiro da lista
00222     if (ant == NULL)
00223     {
00224         aux->nextA = ElimanaTodasAdj(aux->nextA, &inf);
00225         if (*inf == false) return inicio;
00226         inicio = aux->nextV;
00227     }
00228     else
00229     {
00230         // Caso contrário, remove o vértice da lista
00231         ant->nextV = aux->nextV;
00232     }
00233
00234     // Elimina todas as adjacências do vértice e apaga o vértice
00235     ApagarVertice(aux);
00236     *inf = true;
00237     return inicio;
00238 }
```

5.24.3.6 ExisteVertice()

```
bool ExisteVertice (
            Vertice * inicio,
            int id)
```

Verifica a existência de um vértice com um determinado identificador.
Esta função verifica a existência de um vértice com um determinado identificador na lista de vértices. Retorna verdadeiro se o vértice existir, falso caso contrário.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do vértice a ser verificado.

Returns

Retorna verdadeiro se o vértice existir, falso caso contrário.

```
00057 {
00058     Vertice* aux = inicio;
00059
00060     //Corre a lista até encontar o vertice procurado
00061     while (aux)
00062     {
00063         if (aux->id == id)
00064         {
00065             return true;
00066         }
00067
00068         aux = aux->nextV;
00069     }
00070
00071     return false;
00072
00073 }
```

5.24.3.7 InserirVerticeLista()

```
Vertice * InserirVerticeLista (
            Vertice * inicio,
            int id,
            bool * inf)
```

Insere um novo vértice na lista de vértices.
Esta função cria e insere um novo vértice na lista de vértices. Retorna um apontador para o início da lista de vértices e atualiza o valor de inf para indicar o sucesso ou falha na inserção do vértice.

Parameters

inicio	Apontador para o início da lista de vértices.
id	Identificador único do novo vértice.

inf	Apontador para um bool que indica se a execução correu bem.

Returns

Apontador para um booleano que indica o sucesso ou falha na inserção do vértice.

```
00088 {
00089     *inf = false;
00090 
00091     Vertice* novo = CriarVertice(id);
00092 
00093     *inf = false;
00094 
00095 
00096     if (novo == NULL)
00097     {
00098         return inicio;
00099     }
00100 
00101     //Coloca o novo vertice no inicio da lista
00102     if (inicio == NULL)
00103     {
00104         inicio = novo;
00105         *inf = true;
00106         return inicio;
00107     }
00108 
00109     // Se o vertice já existir não cria elimina o vertice criado
00110     if (ExisteVertice(inicio, novo->id))
00111     {
00112         free(novo);
00113         return inicio;
00114     }
00115     else
00116     {
00117         //Ajuda a encontar a posição para inserir o vértice
00118         Vertice* ant = NULL;
00119         Vertice* aux = inicio;
00120 
00121         // Percorre a lista até encontrar um vértice com um id maior que o novo vértice
00122         while (aux != NULL && aux->id < novo->id)
00123         {
00124             ant = aux;
00125             aux = aux->nextV;
00126         }
00127 
00128         // Se o ant for null insere no início da lista
00129         if (ant == NULL)
00130         {
00131             novo->nextV = inicio;
00132             inicio = novo;
00133         }
00134         else
00135         {
00136             // Caso contrário, insere o novo vértice na posição correta
00137             novo->nextV = aux;
00138             ant->nextV = novo;
00139         }
00140 
00141         *inf = true;
00142         return inicio;
00143     }
00144 }
```

5.25 vertices.h

Go to the documentation of this file.

```
00001 
00012 #ifndef VERTICE_H
00013 #define VERTICE_H
00014 
00015 #include "adjacente.h"
00016 
00024 typedef struct Vertice
00025 {
00026     int id;
00027     struct Vertice *nextV;
00028     Adjacente *nextA;
00029 
00030 } Vertice;
```

```
00031
00036 typedef struct VerticeFile
00037 {
00038     int id;
00039
00040 } VerticeFile;
00041
00049 void ApagarVertice(Vertice *v);
00050
00060 Vertice* CriarVertice(int id);
00061
00071 bool ExisteVertice(Vertice *inicio, int id);
00072
00085 Vertice *InserirVerticeLista(Vertice *inicio, int id, bool *inf);
00086
00099 Vertice *ColocaNumaPosicaoLista(Vertice *inicio, int id, bool *inf);
00100
00112 Vertice *EliminarVertice(Vertice *inicio, int id, bool *inf);
00113
00125 Vertice *EliminarTodasAdjacenciasVertice(Vertice *inicio, int id, bool *inf);
00126
00127 #endif
```

5.26 C:/Users/hugoc/OneDrive - Instituto Politécnico do Cávado e do Ave/2023_2024/Estruturas de Dados Avançadas/Dev/TrabII_ESI_↩ EDA_Hugo_Cruz_a23010/src/Main/main.c File Reference

Arquivo principal do programa.

```
#include "libs/InputOutput.h"
#include "libs/caminhos.h"
```

Functions

- int main (int argc, char ∗argv[ ])

  Função princial do programa.

5.26.1 Detailed Description

Arquivo principal do programa.

Author

Hugo Cruz (a23010)

Este programa implementa um grafo orientado e pesado onde utiliza listas de adjac�ncias e listas de v�rtices para cria��o do mesmo.

Version

0.1

Date

2024-05-24

Copyright

Copyright (c) 2024

5.26.2 Function Documentation

5.26.2.1 main()

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

Função princial do programa.

Parameters

argc	Números de argumentos
argv	Array de strings

```
00027 {
00028     setlocale(LC_ALL, "Portugues"); // Configura o idioma Portugues
00029     bool inf;
00030
00031     //Carrega dos ficheiros binário (Primeiro Grafo)
00032     Grafo* g = CarregaDados(argv[1], "v1.bin", "a1.bin");
00033
00034     g = InserirVerticeGrafo(g, 7, &inf);
00035     g = InserirAdjGrafo(g, 7, 1, 54, &inf);
00036
00037     MostraGrafo(g);
00038
00039     GuardaGrafo(g, "v2.bin", "a2.bin");
00040
00041     //Carrga dos ficheiro csv (Segundo Grafo)
00042     Grafo * g2 = CarregaDados(argv[1], "naoexiste.bin", "naoexiste.bin");
00043
00044     bool recebe = ExisteCaminhoGrafo(g2, 1, 3);
00045
00046     g2 = EliminaVerticeGrafo(g2, 3, &inf);
00047
00048     recebe = ExisteCaminhoGrafo(g2, 1, 3);
00049
00050     MostrarCaminho(g2, 1, 4);
00051
00052     int total = DistanciaMinimaEntreVertices(g2, 1, 4);
00053
00054     GuardaGrafo(g2, "v3.bin", "a3.bin");
00055
00056     ApagaGrafo(g2);
00057
00058
00059 }
```

Index